Parallel Simulation Methods for Large-Scale Agent-Based Predator-Prey Systems

A thesis presented in partial fulfilment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Albany
New Zealand

Dara (Minh) Quang Quach

2019

Abstract

The Animat is an agent-based artificial-life model that is suitable for gaining insight into the interactions of autonomous individuals in complex predator-prey systems and the emergent phenomena they may exhibit. Certain dynamics of the model may only be present in large systems, and a large number of agents may be required to compare with macroscopic models. Large systems can be infeasible to simulate on single-core machines due to processing time required. The model can be parallelised to improve the performance; however, reproducing the original model behaviour and retaining the performance gain is not straightforward.

Parallel update strategies and data structures for multi-core CPU and graphical processing units (GPUs) are developed to simulate a typical predator-prey Animat model with improved performance while reproducing the behaviour of the original model. An analysis is presented of the model to identify dependencies and conditions the parallel update strategy must satisfy to retain original model behaviour.

The parallel update strategy for multi-core CPUs is constructed using a spatial domain decomposition approach and supporting data structure. The GPU implementation is developed with a new update strategy that consists of an iterative conflict resolution method and priority number system to simultaneously update many agents with thousands of GPU cores. This update method is supported by a compressed sparse data structure developed to allow for efficient memory transactions.

The performance of the Animat simulation is improved with parallelism and without a change in model behaviour. The simulation usability is considered, and an internal agent definition system using a CUDA device Lambda feature is developed to improve the ease of configuring agents without significant changes to the program and loss of performance.

Acknowledgements

I would like to express my gratitude to my supervisors Dr. Daniel Playne, A/Prof. Chris Scogings and Prof. Ken Hawick for their guidance, patience and wisdom. Without them, this thesis would not have been possible. I would like to acknowledge the members of the Complex Systems and Simulations Group at Massey University for their advice and encouragement. I would like to thank my parents, sister, grandmother and wife for all their continued support. Finally, I would like to acknowledge the financial support I received from the Massey University Doctoral Scholarship.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Artificial Life

Artificial life (ALife) is a field of research where systems displaying behaviours and characteristics of natural life are studied [1]. The main goal of ALife is to model principles or properties of intelligent behaviour and natural life to gain an understanding of life-like phenomena such as self-organisation, self-reproduction, learning, adaptive behaviours, evolution and growth. ALife research occurs through simulations using simulations with computational models, hardware and biochemistry. Example hardware and biochemistry simulations include robots [2] and interacting in-animate molecules [3].

In 1986, Christopher Langton termed the field of research artificial life and organised the first workshop in 1987 at the Los Alamos National Laboratory [4]. The discipline ALife can identify roots from early inventions, research or theories. Automaton is part of ALife history as mechanical machines that incorporate mechanisms to follow a sequence of predetermined instructions in an attempt to capture the form of living things.

The earliest models were perhaps paintings and sculptures constructed with arms and legs to showcase the dynamics of living things as art. Mechanical machines appeared around 135 BC in which Ctesibius of Alexandria constructed a water power mechanical clock composed of early hydraulic technology. During the first century AD, a collection of simple machines in the form of animals and humans produced simple movements powered by pneumatics. It was later during 850 AD that complicated internal dynamics introduced intricate mechanical clockwork which led to the inclusion of life-like motioned animals or figures as a part of the device [1]. In 1735, Jacques de Vaucanson invented an animal automaton in the form of a duck that can mechanically flap the wings, eat and digest food for excretion [5]. The wing alone is constructed with 400 articulated parts. Later in 1737 Vaucanson built a human-like automaton that performed with the tabor and pipe instruments named the flute player [6].

ALife as a computer model exists due to Automata theory as described by Alan Turing in 1936 [7], Turing's construction machine [8] inspired John Von Neumann's self-replicating automata [9]. The invention of computers expanded ALife from mechanics to logic. Von Neumann invented the

Cellular Automata (CA) by a suggestion from his colleague Stanislaw Ulam as an engineering solution was infeasible to construct a universal self-reproductive automaton as shown in Figure 1.1. The artificial machines in Von Neumann's CA model is a group of cells that is regarded as operating as a whole. A machine is interpreted as moving if a specific configuration of the current cells result in another cell appearing in any direction of the Von Neumann neighbourhood. Von Neumann used the CA model to describe a universal constructing machine in which instructions as a collection of coloured cells are used to assemble a new machine. Not only could the artificial machines self replicate but they were thought to reproduce with the capability to pass on assembly instructions which are described by Von Neumann and Banks [10].

The well-known Game of Life is a discrete CA invented by John Horton Conway in 1970 [11]. The game exists on a two-dimensional lattice with rules applying to central cells based on the state of 8 neighbours (Moore neighbourhood). The simple rules, given appropriate starting configurations, resulted in self-organised and emergent patterns that can be grouped as still lifes, oscillators or spaceships [12].

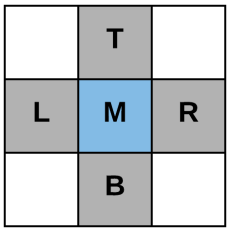

Figure 1.1: Four connected neighbours (*T, L, R, B*) of a Von Neumann discrete cellular automata with 29 possible states for each cell that influences the next state of *M* based on an update rule.

A much simpler version to Von Neumann's universal automata was introduced by Christopher Langton in 1984. The simpler version exhibits a self-reproducing structure in the form of a dynamic loop [13]. These findings inspired the need to understand complex systems resulting in the workshop organised to bring together scientists with research relating to ALife. It was during the five-day workshop in September 1987 at the Los Alamos National Laboratory that a subgroup from the Artificial Intelligence (AI) community began an alliance with biology to research AI as an approach to ALife [14]. This led to studies on intelligent behaviour from the Animat approach, Animal robots and behaviour based self-organising agents [15–17].

In Biochemistry the study of replicating artificial components is perhaps the most ambitious of all the areas of ALife [18]. The evolution of replicating ribonucleic acid (RNA) molecules was exhibited in the work of Spiegelman in 1971 [19]. A transfer of small samples from replicated RNA molecules in a test tube filled with a medium showed mutations in which a moved batch outgrew their competitors by consuming the stock solution at a faster rate.

The simple description of the fundamental levels of structure from natural life is made of at least four stages such as the cellular level, molecular level, organism level and the population-ecosystem level [1]. Understanding the multi-level structures in the field of ALife is achieved with the use of

modelling tools or media such as software, hardware or wet bench lab techniques (wetware).

The software approach is the most widely implemented and popular approach to studying ALife [20]. Compared to the other media mentioned, software models offer the easiest configurable and repeatable system that can produce vast amounts of analytical data at a controlled pace. Moreover, the improvements in computational power put the software approach as alternatives in modelling all four fundamental levels of structure. It should be noted that the software approach does not attempt to simulate real biological systems as a whole, but rather represents an abstract sense of biology where the system is virtual and is a means to investigate properties such as self-replicating, mutating and evolving digital organisms [21,22]. However, adequate knowledge or computer programming is required to develop software-based models; thus ALife research is closely tied to computer science.

Note-able examples of software-based models include Avida and Tierra which models self-replicating computer programs [23, 24], Framstiks: simulation and evolution of 3-dimensional creatures [25], Repast: simulation to investigate development of natural and artificial social structures [26], and StarLogo, Game of Life and Boids to study life-like phenomena from population dynamics to emergent behaviour of complex systems [27–29].

Robots are typically referred to when describing a hardware medium form of ALife. Most hardware-implemented models are for the purpose of studying the organism level. The robots can be in the form of hexapod invertebrates, humans or animals. Robotic simulation of animals can be referred to as hardware Animat [30]. The Animat approach is a bottom-up method designed to reproduce and study the adaptive capabilities of animals and the interactions with their environment while managing their internal autonomous actions [31–33]. As a hardware medium, animals are represented as embodied physical robots that are placed in natural environments, and they consist of three essential components:

1. Sensor - Device that provides information as feed-back from the environment in the form of sensations such as visual, auditory, touch, smell and taste.

2. Actuator - Typically in the form of a motor control device used to mechanically drive the robot to perform physical actions such as crawl, walk, run, jump, climb, swim and fly.

3. Control Unit - Commonly referred to as the brain of the robot. The sensors and actuators are connected to the control unit to await instructions [34]. The control unit may have cognitive features such as memory and learning.

The components used to construct an ALife robot is seen in Figure 1.2. These robots can be simple or complex; simple robots typically consist of simple designs and construction, for example, a simple moving robot that avoids obstacles. Complicated robots typically involve the AI approach of using neural networks to train and learn behaviours based on environmental stimulus [35] such as the six-legged autonomous robot that learns to walk [36, 37].

Hardware-based ALife has many scientific, industrial and military applications, for example, the navigating rat [38], Robirds that mimic the flapping motions of a real bird to fly or a humanoid bipedal robot named Atlas invented by DARPA [39] which is stated to have been developed for the purpose of search and rescue tasks.

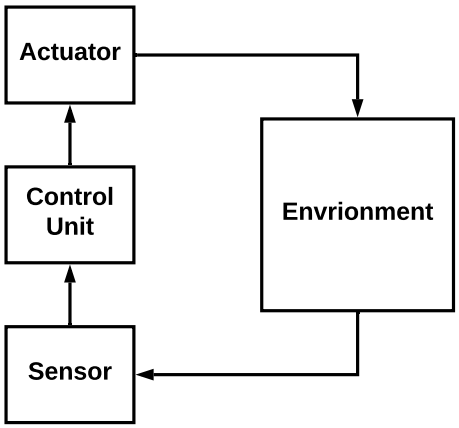

Figure 1.2: Visualisation of the connections between the components of an ALife robot to the environment. Sensors pass situational data to the control unit to instruct the actuators to perform mechanical movements within the environment.

The use of wetware for ALife research at the molecular level produce results that are most similar to natural life. Wetware refers to the molecular approach of scientific lab experiments that typically involve liquid mediums to study behaviours such as self-organisation, self-replication and mutation. Liquid mediums can consists of ingredients such as salts, nutrients, carbohydrates, dyes or amino acids.

Molecular biology and biochemistry are the fields in which the majority of wetware systems are developed. Early examples of ALife wetware were developed to investigate the A-Life principles of the RNA world by synthesising chemical systems at the molecular level [40]. These examples include the work to evolve catalytic RNA from a pool of random-sequence RNA molecules [41] and the use of Darwinian evolutionary mechanisms to allow continuous evolution of molecules in vitro [42]. This type of research has real applications such as the techniques of directed molecular evolution used by the pharmaceutical industry to research substances for medical application.

1.2 Agent-Based Models

Agent-based modelling (ABM) is a widely accepted approach to simulate complex-systems at the microscopic level computationally. Autonomous agents or individuals that interact are vital components of an ABM simulation. Agents behave based on simple deterministic rules or abstract and stochastic representations of stimulus-response mappings [43]. Agent behaviour that requires interaction may result in emergent phenomena such as patterns, structures and system behaviours [44].

Complex adaptive systems (CAS) considered as a historical root of ABM due to the focus on how complex adaptive behaviour can emerge in nature as a result of interaction between autonomous agents [45]. A note-able contribution to ABM is John Holland's work in which he identifies and categorises the principles and mechanisms of a CAS [46].

The concept of autonomous interacting agents was described by John Conway and Thomas

Schelling with the use of graph paper and coins. Conway's Game of Life investigated the emergent patterns of CA producing a publication in 1970 signifying emergent patterns as a result of specific starting configurations [11] while Schelling's simulations exhibited exaggerated separation and patterning as a result of dynamic movement of agents in 1971 [47].

In 1980 Robert Axelrod made use of ABM concepts by way of individual participants in his Prisoner's Dilemma game [48, 49] and furthered studies in social sciences by using ABMs to investigate artificial societies [50] and social influences [51]. Axelrod elaborates on the use of ABM in a publication in 1997 [52].

With the improvement of technology during the years of CA research, new techniques were employed to develop more advance ABM simulations in the case of Boids by Craig Reynolds 1986. Reynolds then published the findings of his model as he presented emergent system behaviour that exhibited realistic flocking behaviour of birds, herding of land animals and schooling of fish [29]. What makes the Boids ABM interesting is the absence of central control during a large number of agent interactions.

The interest of ABM stemmed from social science and ecology but can be found in other fields such as micro-biology. A wide range of examples includes predator-prey models [53–56], forces and evacuation of crowds [57–59], simulations of multicellular behaviours in bacteria [60, 61] and Flocking [29, 62] .

The practice of ABM started as techniques and tools to implement complex adaptive systems using a computer. The increase in computational power expanded the limits of ABM and provided an extra means to explore non-linear system dynamics [50]. Reynold's work showed even simple agents are capable of exhibiting complex macroscopic behaviour [29]. The key concepts of ABM are well described in a tutorial by Macal and North [63] and explains that the typical structure of an ABM consists of three aspects:

- Multiple Autonomous Agents - Agents have an internal state consisting of multiple attributes that describe their condition at a specific time. For example, a simple agents state may require a position and energy attribute to identify a location and enumerate its potential.

- Behaviours and Interactions - Agents have a set of behaviours that define how an agent reacts to changes in their situation. Behaviours typically include rules that allow interaction between agents and the environment. For example, an agent may consume another to replenish internal attributes such as energy.

- Environment- The habitat, domain or world in which agents occupy and exist. The environment may serve to supply information on the locality of agents or host global attributes to represent resources constrained by spatial locality such as grass that only grows in certain locations within the environment.

The term agent is loosely used to represent entities such as agents in software, agents in hardware as robots or agents in the economy. The definition of an agent in ABM can vary based on their domain and purpose; however, they share basic common concepts such as autonomy and identity in

the form of attributes to distinguish individuals. The literature from Jennings [64] provides the description of common characteristics from a computer science point of view and expands on essentials described by Macal and North [63].

Common characteristics of agents are:

- State - Attributes that may vary over time and describes the agent's situation.
- Identity - Attributes that uniquely distinguishes an agent from other agents.
- Interaction - Dynamic interactions with other agents or environment which influences behaviour.
- Autonomy - Self-directed and independently functioning.
- Purpose - Objectives as a purpose they try to achieve.
- Learning - The ability to learn and adapt to problem solve.

Although Jennings describes purpose and learning as a necessity, Macal and North categorises these characteristic as useful. Interactions of multiple agents and individual behaviours are examined closely as these are influential properties that may produce macro-level emergence [65]. Agent Interactions can be described as agent relationships and connectedness. The notion of agent relationships is specified as what or who agents can form a connection with, such as the case of an altruistic predator-prey model in which agents share resources with their neighbours [66].

The type of interactions is specified based on the purpose of the ABM. The agents may interact based on the goal and type of world in which they are situated. For example, on lattice type environments the Von Neumann CA uses a Von Neumann neighbourhood connectivity whereas Conway's Game of Life uses Moore's neighbourhood connections.

Neighbourhood communication can be considered social interactions and can occur on other environmental topologies such as networks, or continuous space in two or three dimensions shown in Figure 1.3. Examples of ABMs with various types of environment topology include the use of a lattice to model molecular self-assembly [67], continuous-space environments for the flow of land and air traffic [68, 69] and network type models to simulate social networks [70]. Geographical information system (GIS) are becoming more common, and examples include the use of spatial geographic information to implement an ABM [71] or a model that produces spatial patterns of tree mortality as a result of interactions with the mountain pine beetle [72].

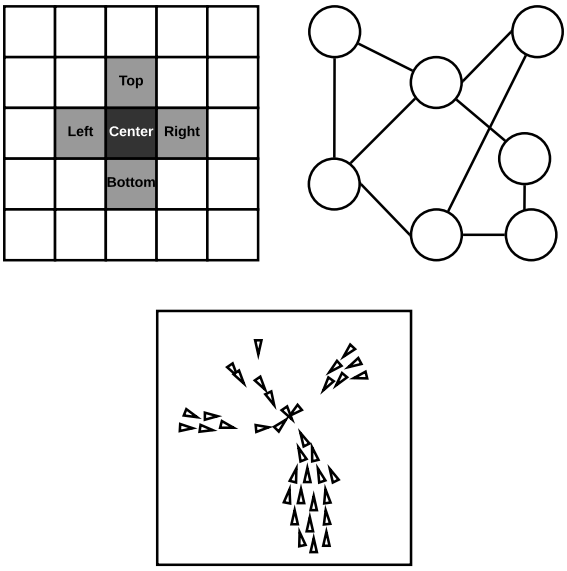

Figure 1.3: Topologies of spatial agent-based models: lattice, network, euclidian space.

The application of an ABM can be implemented with development on computational devices using object-oriented languages programming languages such as C++ or Java. These type of languages are appropriate for implementation as objects and agents share similar concepts [73]. However, writing custom ABM software requires a level of programming skills and knowledge that is sufficient to implement a functioning and optimised simulation.

There is a wide range of software platforms to assists the process for developing ABMs, these include toolkits such as MASON [74], SWARM [75], NetLogo [76,77], FLAME GPU [78], RePast [26], Framsticks [25, 79] and AnyLogic [80]. The primary domain of these frameworks may vary, for example the frameworks NetLogo and RePast target social sciences whereas SWARM and AnyLogic for general purpose and Framsticks for artificial life. There is a comparison of these packages in the literature by Railsback et al [81] and Alan Isaac's refinement of the introduced template models for ABM [82]. Moreover, some issues in regards to ease of use, design and flexibility are introduced.

The ABM approach has often been used to represent Equation-based predator-prey models on the micro-level. Predator-prey ABMs provide a means to understand the processes that lead to the emergent behaviours of the model. The following section introduces the predator-prey models and the connections between the models on macro and micro-level.

1.3 Predator-Prey Models

The dynamics of predator and prey interaction is a common topic in the field of ecology. In a predator-prey model, species may progress their existence by competing, evolving or propagating in an attempt to consume resources and reproduce. The competitive nature of population dynamics in predator-prey models help to extend the application to other fields such as biology and economics.

Depending on the application of the model, the coupled species may take the form of a herbivore, carnivore, tumour cell or consumer in business ecosystems. Predator-prey models can be simulated at the macro or micro level using linear and non-linear differential equations and agent-based modelling.

The origins of predator-prey models can be traced to the Verhulst logistics equation introduced in 1838 [83] which is a mathematical model based on Thomas Malthus's literature on the principle population dynamics in 1798 [84]. The logistic equation is a basis of which other single-species population dynamics model are generalised from such as the Lotka-Volterra predator-prey model proposed in 1910 [85]. The extension to a predator-prey model occurred in 1920 when Alfred Lotka used an example of a plant and an animal to represent predator and prey. The Lotka-Volterra equations were used to model the dynamics of predator-prey interactions in 1925 [86] and soon after in 1926, Vito Volterra independently investigated the predator-prey equations for two or more associated species [87].

The evolution of the predator-prey equation is detailed in Alan Berryman's publication in which the significant contributions to the equation over the 19th century is discussed [88]. For example, Berryman stated the original Lotka-Volterra equations allows the prey population to grow indefinitely when there are no predators and can be corrected by adding a logistic term to the prey. Another extension includes a functional response which is the intake rate of a consumer as a function of food density [89], the equations used in ecology are known as Holling type I, II and III.

In 1948 Leslie expanded the logistic predator equation by using ratios rather than products to focus on the community equilibrium [90]. Berryman also extends the ratio-dependent functional response additions by Soloman [91] and also Arditi and Ginzburg [92] by stating that the equations should be derived as per-capita rates of change [88].

The Lotka-Volterra coupled equations is shown in Equations 1.1 where x is the population number of prey and y is the population number of predators. The parameters α, β, δ and γ are positive numbers that describe the population interaction of the species. To represent the interactions the change in prey population numbers caused by the growth αx minus the rate it is preyed upon by predators βxy and in the predator equation the change in population numbers is a result of the growth in predator population δxy minus natural death $-\gamma y$.

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= -\gamma y + \delta xy \end{aligned} \tag{1.1}$$

If the assumption that prey has an unlimited food source and the only food source of the predators is prey, then if there are no predators prey would grow exponentially:

$$\frac{dx}{dt} = \alpha x \tag{1.2}$$

and without prey, the predator population would collapse:

$$\frac{dx}{dt} = -\gamma y \tag{1.3}$$

The predator-prey equations by Lotka and Volterra have been shown to explain the observations of an anomaly of predatory fish populations increasing during 1914 to 1918 [93] and the dynamics of natural predator-prey populations provided by the data of the lynx and hare in their natural habitat [94].

Independently extensions to the Lotka-Volterra equation in other disciplines include the Kermack-McKendrick Model for susceptible-infective interactions in epidemiology [95], the formation of chemical radicals during the combustion of H_2O_2 in combustion theory [96] and the Palomba's Model regarding the economic relation of consumption and capital goods in economy [97].

The parameters of the equations can be perturbed to understand the resulting effect on the population of the species over time. There are studied variants to the model in which the differential equations are further extended or modified to explore system dynamics such as simulations of a spatial Lotka-Volterra model [98], a spatial model of multiple species [99], complex feeding chains [100] and species survivability and altitude dependence [101]. The analysis of these models concludes that many slightly perturbed factors may impact the model resulting in uncertainty in the population dynamics. Figure 1.4 shows examples of simulation screen captures of the spatial Lotka-Volterra model experimented with standard two species, multiple species and two species on a digitally elevated system.

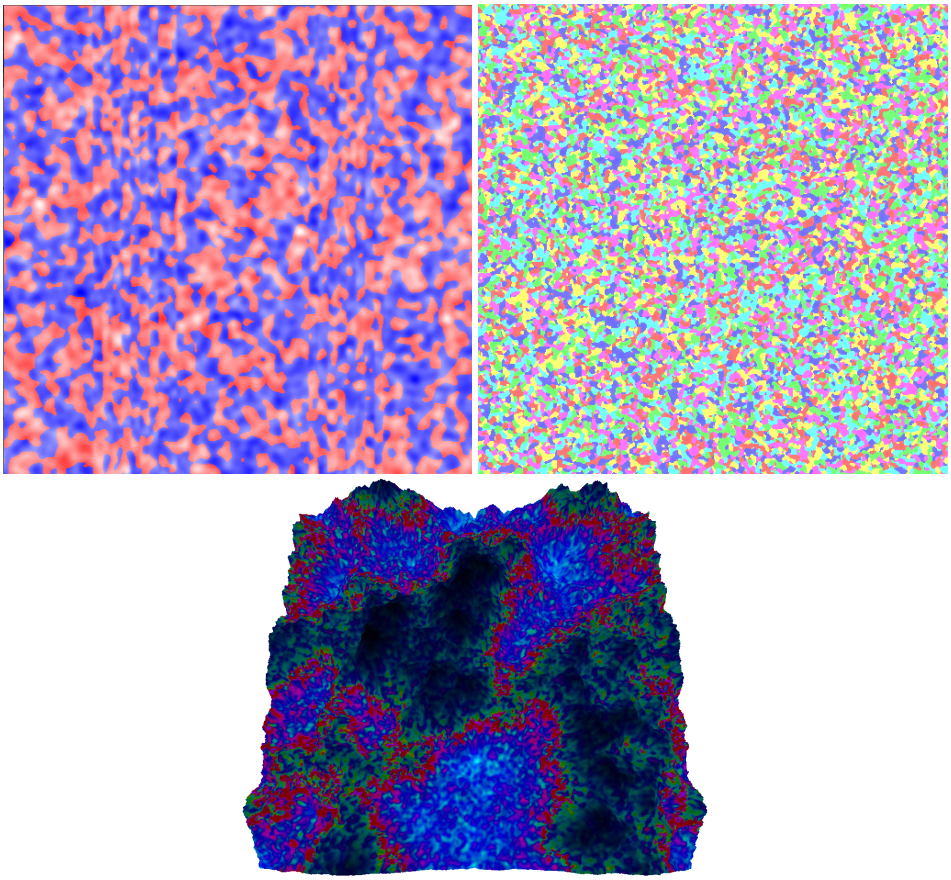

Figure 1.4: Example screen captures of spatial Lotka-Volterra simulations with: two species (left), multiple species (right) and two species on a digitally elevated model (bottom).

Although many equation based predator-prey models exist, it is widely accepted that they investigate phenomena at the macroscopic level. Interesting phenomena such as the periodic boom-bust cycles of population levels that often appear in Lotka-Volterra based models naturally lead to a desire for understanding these behaviours at the microscopic level.

The attempts to model micro-level behaviour have resulted in the use of the bottom-up approach which species are represented as a group of individuals or agents. Agents of a predator-prey model typically consist of individuals that have behaviours and characteristics of the species they belong to. For example, Predators naturally hunt and eat prey and Prey graze as herbivores. Predator-prey agent-based models typically include fundamental behaviours such as reproduction, movement and predation that may result in periods of population boom-bust cycles and emergent phenomena that may not be seen in macro-level models.

A common way to compare micro and macro level models is to analyse the population dynamics. Micro predator-prey models are typically implemented with the individual or ABM approach to compare with equation-based models using the resulting dynamics in the population data produced by the simulations.

Specifically, researchers attempt to make a connection between the models using the result of the periodic boom-bust cycles of population levels exhibited by both approaches. For example, some studies use the spatial agent-based approach to model predator-prey interactions resulting in similar population dynamics while producing complex emergent behaviour for classification and quantification [53]. Other simpler implementations include the population-driven individual-based (PDIB) model developed to focus the investigation into the effects of spatialising the predation interaction on population dynamics [102].

A comparison of predator-prey interactions from the perspective of micro and macro level approaches have been presented in a study [103] which concludes the individual-based approach is acceptable to construct more accurate models of population dynamics. The investigation used a discrete Markov model on predator-prey interactions for the micro level perspective to show similarities to the Holling type II predator system [104] while maintaining a globally stable equilibrium. Other examples include examining the parameters to implement and compare and individual-based system to the Lotka-Volterra model using the Bullant programming language [105].

Model size in terms of topology and the carrying capacity of agents may also impact the emergent phenomena and population rise and fall cycles. For example, the Animat predator-prey model used in this dissertation requires a lattice environment size of 200 for the simulation of the system to continue without immediately crashing to zero. There is not much investigation into computationally modelling large model sizes of hundreds of thousands to millions of agents which requires a significant amount of processing power and is often a limitation for many other models. Assumptions such as large scale models may smooth out the fluctuations of population numbers caused by individuals, or a collective of individual behaviour can be made from existing models. However, these assumptions require the models to be extended to a larger-scale to provide new data for investigation.

A particular Agent-based model that can simulate predator-prey interaction is the ALife Animat model. This model extends the typical discrete space ABM topology by allowing many agents to occupy a cell in a lattice environment. This can lead to large population numbers that may produce emergent system behaviour as a result of local agent interactions.

1.4 Agent-Based Animat Model

The Animat model is an ALife approach to studying behaviours from live systems in computer simulations [32]. Avida [106] and Tierra [107] simulators were early models using the Animat approach to study the complexities of life that result in the exhibition of life-like behaviours. The Animat model used in this thesis is an agent-based computer simulation initially developed to study microscopic level predator-prey systems and capture the boom-bust mechanics of macroscopic partial differential equations [53, 108].

Early developments of the model led to interesting emergent behaviour or complex phenomena which is an important aspect of artificial life. The particular interest of emergent phenomena from these models is that they are not explicitly programmed to exhibit behaviours of the system as a whole. Rather the individuals are programmed with simple rules to exist and behave according to

their situation and environment. Simple rules that lead to interaction between agents may result in emerging behaviour such as the locality of agents to form spatial patterns that resemble defensive spirals [109], spontaneous emergence of spatial tribes [110], directional waves and transitional clusters [53]. Examples of these emergent life patterns are shown in Figure 1.5 as captures of the Animat simulation during the standard density phase reaching system carrying capacity. Other emergent phenomena exhibited by the Animat model include migration herding behaviour in resource-scarce environments [111], subspecies segregation and spatial pattern growth [112, 113].

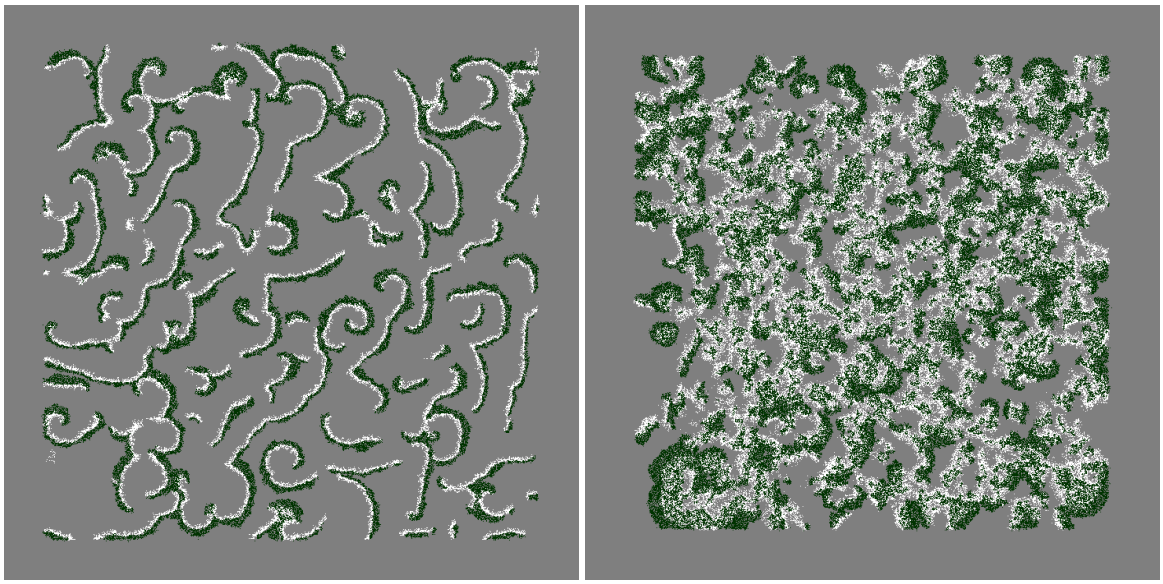

Figure 1.5: Screen capture of Animat simulation using standard and ranged search strategies during system carrying capacity; where prey are coloured green and predators white.

The modern Animat model is an agent-based predator-prey simulation that consists of predator and prey agents that exist in discretised space or cells. Agents can freely move between each cell and more than one agent can occupy a cell. Predator and prey agents have typical behaviours such as moving, breeding, grazing and eating. Prey agents graze and predator agents hunt and eat prey. In this Animat model a prey can only be eaten by a predator. The details of the Animat model is discussed in Chapter 3.

This Animat model is robust and can be extended to explore specific animal behaviours such as altruism [66], hunting cooperation [114], camouflage [115], evasion [116] and adaptive resource trading [117]. These experiments typically use environments that result in model sizes of a few hundred thousand agents. Exploring these with larger model sizes are restricted by the computational power of a single-core processor when the simulation is implemented to execute serially. This may cause tests and complicated additions to the model to be time-wise infeasible to collect data or analyse in real-time.

1.5 Parallel Computing

Parallel computing is the use of processing units to perform computation on a task or tasks simultaneously. A computational problem can be decomposed into smaller tasks for parallel execution to improve performance [118]. Parallel computing has gained and continues to gain interest as the philosophy of Moore's Law [119,120] no longer holds due to the physical constraints of achieving higher frequencies in modern processing units [121–123]. Parallel computation can be achieved on various hardware and architectures from modern mobile phones and desktops with multi-core processors, clusters with many single or multi-core units to distributed systems.

In 1955 the first commercial computer with transistors called the IBM 704 was introduced by International Business Machines (IBM) led by Gene Amdahl [124]. Soon after the in 1958 John Cocke and Daniel Slotnick discuss the use of parallelism in numerical calculations and as a result, Slotnick later proposed the design for SOLOMON, a single instruction, multiple data (SIMD machine which serves as a starting point for parallel computing machines [125]. The first parallel machines were built with multiple instructions, multiple data (MIMD) multiprocessors named D825 and introduced by Burroughs in 1962 [126].

The interest in parallel computation machines emerged in the 1970s during which (SIMD) machines were developed such as the Illiac-IV [127]. Examples of other multiprocessors include the PDP-6/KA10 [128], El'brus [129] and C.mmp [130]. It was during that decade that the literature on parallel computing emerged, such as the studies of massive bit-level parallelism [129, 131], description of shuffle network as a basis for later work on parallel computer topologies [132] and the parallelism in random access machines model for complexity analysis of parallel algorithms [133].

Over the years single core processing units have been phased out due to the issues of power consumption and heat generation problems of modern transistors. This led commercial manufacturers such as Intel and AMD to shift towards multi-core technology in 2005 as the yearly performance increase of processors dropped [134]. The purpose of the shift is the idea that multiple lower frequency energy-efficient cores improve overall performance by computing tasks simultaneously.

Modern consumer desktop computers consist of 6 to 8 core CPUs that have varying frequency ranges and number of threads. An example modern chip is the Intel i9-9900K CPU that are constructed with cores capable of running at higher than standard clock frequencies, reaching 5 GHz.

Modern chips such as the Intel i9-9900K include controlled overclocking that involves some of the multiple cores to operate at a lower frequency in order to meet the energy demands of the cores with increased clock rates. AMD offers a 32 core processor named the Ryzen Threadripper 2990WX; however, they are actually 4 sets of 8 core chips joined together that can be boosted to clocks of 4.2 GHz from their 3GHz base clock.

The Graphics Processing Unit (GPU) hardware advanced alongside the development of multi-core systems. GPUs can be referred to as many-core circuits or cards that are dedicated or connected via the Peripheral Component Interconnect (PCI), Accelerated Graphics Port or PCI express expansion slots of the motherboard. GPUs were initially developed for accelerating graphical processes such as rendering computer game graphics or image processing. They are more suited to these tasks than CPUs due to their massively parallel architecture that is capable of processing vast amounts of

data simultaneously.

The demand of the computer game industry has led to the development of very powerful GPU accelerators of today such as the RTX 2080 Titan consisting of 72 streaming multiprocessors capable of utilising 24GBs of GDDR6 memory with 672 GB per second bandwidth. GPUs are designed for massive parallelism capable of many teraflops of floating point operations per second which a modern equivalent CPU cannot compete with. The high performance of GPUs has led to general purpose use commonly referred to as GPGPU where the shader cores can be used to process multiple streams of data simultaneously.

The use of GPU has continued to gain traction in scientific computing and commercial endeavours due to the massive parallelism and computational power offered by these architectures [135]. The general computing nature of GPUs has enabled significant increases of parallelism to process tasks traditionally performed on a CPU such as the computation of numerical models with a vast amount of speed-up [136], the reconstruction of medical imaging [137] or the mining of cryptocurrencies such as bitcoins [138, 139].

The attraction of GPGPUs is its relatively low cost for the performance gained. GPUs continued to outperform commonly compared architectures that cost considerately more such as Intel's many integrated core (MIC) architecture consisting of Xeon Phi CPUs. When compared to the Xeon Phi the GPU is more efficient in cases of irregular memory access, and atomic operations [140] and has a higher per year percentage of increase in floating point operations per second(FLOPS). Further support of GPUs in high-performance computing is shown with them appearing in 3 of the top 10 supercomputers in the world such as the IBM Summit which contains 27,648 NVIDIA Volta GV100 GPUs with 6 per node that is the worlds fastest supercomputer capable of performing 200 PFLOPS [141].

High-performance computing may extend the parallelism provided by the common setup of a CPU machine with a GPU accelerator using some form of connection to many CPU machines with many GPU accelerators such as the case with IBM SUMMIT. Distributed parallel systems are similar and are generally categorised by how it is connected, what it is connected with or how it shares resources [142]. They share the concept that they are connected for the purpose of processing data or tasks simultaneously.

The types of parallel computers with multiple or many connected nodes can be categorised as follows:

- Cluster - A type of distributed parallel computing that has low latency due to connections with specialised hardware and high-performance networks. Each node of a cluster is connected via a local area network and are set to perform the same task that is software scheduled and controlled.

- Distributed - A system where servers of parallel computing systems on different networks are connected to process a certain task by sharing resources. Distributed parallel computer systems communicate using message passing interfaces and may have latency and unexpected failure issues.

- Grid and Cloud - A type of distributed system where widely distributed computers share a large amount of resource for access by entities no matter where they are located.

An example of a connected cluster is shown in Figure 1.6, each node is connected to a network switch and may contain one or more processors that have multiple cores a client computer can connect to.

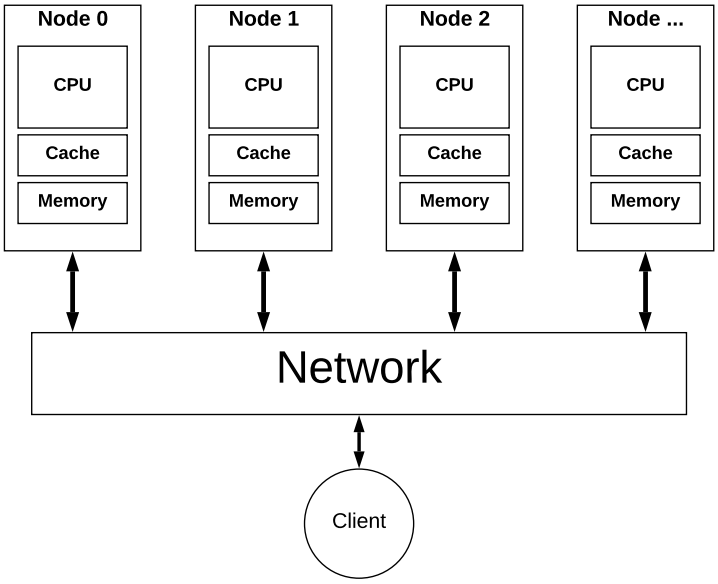

Figure 1.6: Visualisation of a computer cluster.

Programming for parallel computers is still not a trivial task. Throughout the history of computing, programs are generally written to run sequentially as this was simpler with the assumption of Moore's law philosophy remaining true. The development of single-core architectures has been stalled in favour of designing multi-core CPUs by two major manufacturers Intel and AMD, and exploiting parallelism in multi-core systems require the program to be explicitly written to make use two or more cores. The challenge in making use of parallelism is the process of porting sequential programs to parallel architectures. Different parallel architectures have different configurations of memory and processing units; modern architectures have different types of memory and levels of cache. Writing programs to exploit these architectures is not a simple task as there is the requirement of understanding the architecture and memory design to deal with dependencies and control execution flow. The multi-core and GPU architectures are introduced to support the parallel implementation details of this thesis.

The typical architecture found in modern processing units are Multi-core CPUs, Figure 1.7 shows a traditional configuration of a four-core processor architecture. Each core of the processor has individual private L1 caches, and all share the L2 cache. The L1 cache is bound to the core and is frequently accessed which requires the cache to be low latency with fast throughput, L1 is much faster than L2 albeit more expensive to construct.

The L2 cache exists to manage the situation of L1 caches misses and can be larger due to fewer requirements such as higher latency and lower number of read and write ports. L2 cache is critical for performance as the non-existence of L2 would require L1 misses to be handled by main memory which has lower bandwidth. Modern Intel multi-core CPUs include an L3 cache which acts as the last level shared cache and allows each core to have a private L2 cache or the case of AMD bulldozer the L2 cache is shared between two cores. The L3 cache acts as the backstop for cache coherent cases which may otherwise require access to the main memory.

A cache coherency problem is a situation where multiple cores have access to a shared memory location, and one of the cores modifies the data at that address which may result in an incoherent cache which is when other cores have a different data values at that memory location. Cache coherency remains a problem for multi-core chipmakers.

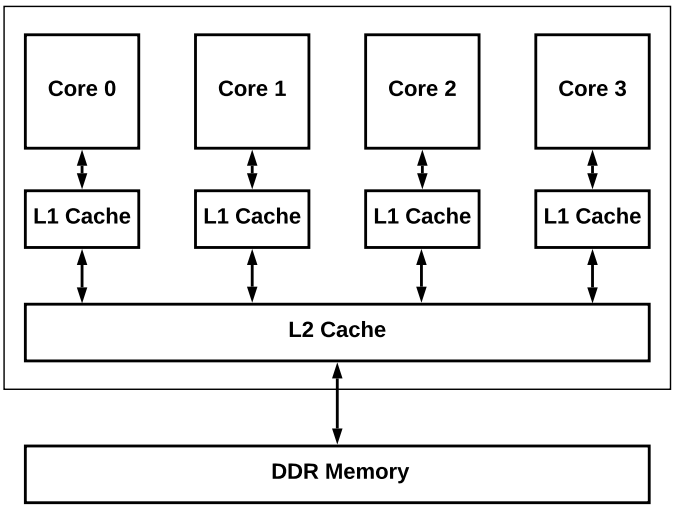

Figure 1.7: Traditional multi-core processor with four cores and individual L1 caches, L2 cache is shared with the processor connected to DDR memory via a bus interface.

NVIDIAs Turing architecture is the latest development for the RTX line of GPUs that is based on the Volta microarchitecture. The RTX Turing GPUs include consumer AI Tensor cores and real-time ray tracing processors. The Ti and Titan versions are constructed with 4352, and 4608 CUDA cores accompanied by 11GBs and 24GBs of DDR6 memory. CUDA is a parallel computing platform developed by NVIDIA for general purpose computing using their graphics cards. Figure 1.8 visualises the Turing streaming multi-processor architecture with specialised INT32 units, FP32 units, Tensor cores and Ray tracing cores that can be programmed on CUDA and has 7.5 compute capability. Each series of architectures are designed differently and have different compute capabilities, specialised cores such as the Tensor and Ray tracing processors are only introduced in the Turing architecture and do not appear in the Pascal, Maxwell, Kepler or older series.

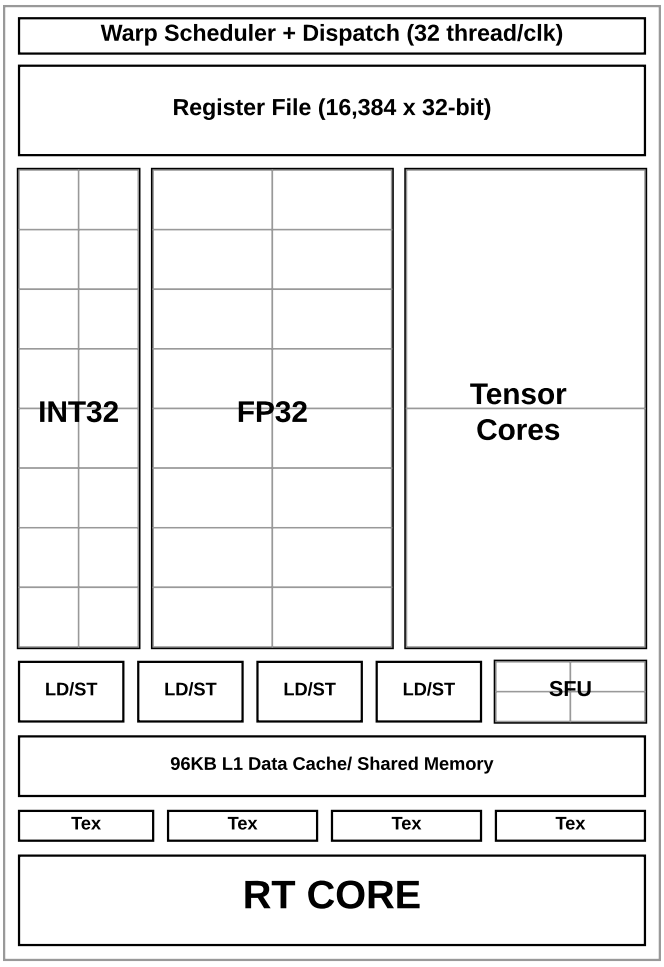

Figure 1.8: Visualisation of the Turing streaming multi-processor architecture.

GPUs have several types of memory for various purposes with some shared between threads of a multi-processor and some types shared by all multi-processors. Although new architectures have higher bandwidth memory, it is still vital to select the type of memory suited to the problem to achieve high performance on GPUs. The different types of GPU memory are listed as follows:

- Global Memory - Is accessible to all threads and can be written or read from the host (CPU). It is the largest and also the slowest type of memory on the GPU device. Ensuring global memory accesses are coalesced and minimising redundant accesses where possible can help to improve the performance of global memory use on all GPUs. A coalesced memory transaction means all threads in a warp access consecutive memory addresses thus allowing the transaction to be a single load and store operation.

- Shared Memory - Type of memory shared by threads of a multi-processor which is on-chip allowing it to be faster but smaller in size compared to global memory. Each SM of the Turing

architecture has 64KB + 32KB or 32KB + 64KB of configurable L1/Shared memory. Shared memory is allocated per block and threads can access data loaded from global memory by other threads in the same block. Data that requires repeated read and write instructions by threads of a block can be loaded into shared memory for faster processing before being written back to global memory [143, 144].

- Registers - The fastest type of memory available on the device that is allocated to each thread to store variables. Turing architectures offer 255 registers per thread and when register resources are used up, data spills into the L1 cache. Register spilling is not a performance issue if the spilling is contained on the on-chip L1 cache. Further spills to device memory may affect performance, and design techniques such as removing registers for values that can be easily calculated or compacting smaller values into a single type can reduce the number of registers required.

- L1 and L2 cache - Provides load and store services to SM multiprocessors with the L1 on-chip and L2 off-chip. L1 uses the same location as shared memory and can be configured as previously mentioned. L1 and L2 can be used for register spilling and the L2 the cache ranges from 3000 to 6000 KB on Turing architectures.

- Local Memory - Resides in global memory; thus local memory accesses have the same high latency and low bandwidth as global memory accesses. Local memory also requires coalesced transactions for better performance and are used by the compiler to place large arrays or variables that consume too much register space, as previously mentioned (register spilling).

- Constant Memory - Resides in global memory space and is cached in the constant cache. Constant memory may not be modified from the device so constant variables are set from the host prior to launch.

- Texture Memory - Read-only memory that is cached on-chip like constant memory and may provide higher effective bandwidth by reducing memory requests to global memory. Cached texture memory is fast when there is a cache hit otherwise a texture fetch costs one global memory read.

Exploiting multiple cores and shared resources to parallelise serial programs for performance improvements is becoming common. Implementing parallelism is not straight forward as the process involves coordinating multiple cores to solve a single problem which may require addressing issues and challenges such as introduced bugs, race conditions, deadlocks or resource contention. The challenge that may occur may depend on the architecture, and some issues are described as follows:

- Synchronisation - Management of threads by coordination with synchronisation barriers such as locks or semaphores that make use of the mutual exclusion primitive called a mutex. Low-level thread management typically requires the coordination of locking mechanisms to be programmed and in such cases may result in extra overhead of too many lock operations which can negate the performance improvements provided by parallelism.

- Occupancy - Oversubscription can occur in programs written for multi-core architectures when tasks are decomposed to small jobs that require more threads than there are cores available. This can be an issue when too many threads are competing for processing time on multiple cores. Undersubscription is also an issue when the overhead to launch threads is more expensive than processing the tasks.
- Task Scheduling - Scheduling is of importance when the size or priority of tasks is a factor. If tasks are not scheduled in a sensible order, then the total time taken to process all tasks may take longer than required. This is typically a problem for multi-core architectures with a low number of available cores.
- Resource contention - Contention occurs when threads or cores compete over shared resources such as memory or cache. Many threads that share a cache of a single core may displace each threads data in the cache; this is not a problem if all threads assigned to a core requires the same data.

The various types of parallel architectures have different libraries to ease the process of developing parallel programs on computer languages by automatically addressing some of these issues. These libraries provide different levels of thread control such as POSIX threads for low level and Open Multi-Processing (OpenMP) for high-level design. These parallel CPU programming libraries are described and compared for implementation of the multi-core Animat model in this thesis in Chapter 4. The CUDA programming platform and model is used to Implement the Animat model on a GPU in this thesis due to better performances than OpenCL in kernel execution and data transfer between the host and device memory [145]. CUDA template libraries such as Thrust and CUDA UnBound (CUB) can be used to support the development of the implementation and are discussed in Chapter 5.

In parallel computing, Amdahl's law is commonly used to predict speed up of programs when using multi-core or many-core architectures [145–147]. Amdahl's law states that latency can be calculated based on the sequential portions and parallel portion of a program. The parallel portion is divisible by the number of cores utilised, and if all parts of the program must be executed sequentially then, parallel architectures will not increase performance. Programs that have a greater parallel portion can be considered embarrassingly parallel as the parallel portions of these programs can be efficiently processed by multiple or many cores of the architecture.

Modern computational devices all use multi-core or many-core architectures. The future of programming trends towards parallelism with modern examples of real-world applications using parallel computing to solve problems that can be parallelised such as medical imaging, mining cryptocurrencies, AI neural networks and scientific computing. Multi-core CPU and GPUs are increasingly used in medical imaging to provide improved health care efficiently which can be complicated by the magnitude of the data acquired [148–150]. The power of GPUs is commonly used to mine cryptocurrencies such as Bitcoin due to the low cost and high performance gained over traditional CPU systems [151]. GPUs are often used to increase the performance of models in scientific computing and neural-networks for applications such as deep learning. There have been investigations into the use of GPUs for these fields [135, 152–154].

Parallel computing is a common approach to improve the performance of ABM simulations. GPUs are increasingly used to compute ABMs at a large-scale at which serial implementations struggle to simulate in a feasible time-frame. The following section discusses some parallel methods and frameworks that are used to implement ABMs on the GPU. This helps to identify the uncommon problems when implementing the Animat ABM on the GPU.

1.6 Parallel Methods and Frameworks

In Agent-Based Modelling (ABM) the term large-scale refers to both the size and complexity of the simulation. Complexity is inherent in ABMs as they typically consist of agents that are dynamic, autonomous, heterogenous and interactive. ABMs have increased in complexity in the last 50 years from simple cellular automata such as the game of life devised in 1970 [11] to the recent use of an agent-based transport simulation (MATSim) in which agents as cars on a connected network of queues optimise their route using the co-evolutionary principle [155].

Serial implementation of ABMs are limited in computational power to simulate large agent populations in a feasible time-frame. Parallel implementations have been developed to simulate ABMs using larger population sizes of millions of agents. However, implementing ABMs on parallel architectures is challenging as the parallel methods used may need to consider race conditions when computing parts of the simulation simultaneously. Some frameworks have been developed to assist the process of implementing ABMs on parallel architectures such as multi-core CPU, CPU clusters and GPUs. Examples of these frameworks include include HLogo [156], OpenABL [157] and FLAME GPU [78].

There has been on-going research into using the GPU for ABMs due to the potential performance gains and accessibility when compared to CPU clusters. Generally the common goal is to update all agents in parallel. The GPUs fine-grained parallelisation model is well-suited to this problem as the agents can be fine-grained tasks that can be computed in parallel. This section discusses notable work on ABMs implemented on GPUs. A review of the parallel methods and frameworks are provided and the problems they address are discussed. Issues relevant to parallelisation of the Animat model on the GPU are identified.

1.6.1 Non-Conflicting Agent-Based Models

Typically, ABMs can be categorised based on the environment topology such as discrete or continuous space. Discrete space can be represented as cells of a lattice that may have neighbouring cells. Continuous space has no notion of cells or discrete regions. Agents store their locations as coordinates for each simulation step.

Examining the rules of the ABMs is of particular importance for parallelism on the GPU and can be classified as conflicting and non-conflicting based on the agents and the rules of the model. Conflicting ABMs consists of situations in which two or more agents attempt to perform an action that affects a unique resource and only one agent is allowed to perform that action. An example of a conflict is when two or more agents simultaneously attempt to move into a particular cell. The

model rule can be that only one agent is permitted to be in a cell at anytime. Non-conflicting ABMs consists of agents with actions that do not depend on the actions of other agents when performed. For example, all food sources may be unlimited for agents so there is no need to consider distribution.

Conflicting ABMs are not a problem for serial implementations as a sequential order of update for agents will simply resolve a conflict such as movement. The first agent to be updated moves into a particular cell, the agent after acknowledges the fact and may decide to move somewhere else. This is a problem when considering parallelisation of the ABM. All agents updated simultaneously would require some process of resolving conflicts.

A list of notable ABMs with agents that do not have conflicting behaviours are as follows:

- Game of Life - Cellular automata of agents in discrete space that may live or die depending on the rules and the state of neighbouring agents.
- Boids - Continuous space model with agent that calculate their final velocity and direction based on factors such as separation, alignment and cohesion relative to near by agents.
- Crowd Simulations - Continuous space model that uses boids like separation and alignment for collision avoidance.
- Ant Colony Optimisation - Graph model that optimises problems such as the travelling salesman based on the behaviour of real ants exploring a path between their colony and a source of food i.e new paths or solutions by each ant following a pheromone trail to leave a new one.

These type of ABMs are not particularly difficult to port a serial implementation onto the GPU. Typically these ABMs can be updated based on two phases, the current state and the future state. The future state of agents can be simultaneously computed based on the current state. This is straightforward as there is no dependencies between agents in the current state.

Typically, research for Non-Conflicting ABMs often focus on optimisations to improve the simulation performance on GPUs. For example, some methods of spatial partitioning have been included to improve memory access on the GPU by minimising memory accesses for irrelevant data such an agent that is not in the vision of another. One example is using a tree data structure such as an Octree for continuous space models in which agents can be stored as nodes [158]. The benefit is the ease with which the nodes containing other agents not within vision range is skipped. Another approach is the use of environment partitioning in which the agents within each partition can be updated efficiently by making use of specific GPU memory [159]. These optimisations have been developed for continuous space and may not suit discretised space ABMs such as the Animat.

1.6.2 Conflicting Agent-Based Models

Implementing ABMs onto the GPU with agents that may cause conflicts when updated simultaneously is non-trivial. It can be challenging to develop methods to resolve the conflicts with improved performance on the GPU. Ideally the conflict resolution methods should not change the model or be unbiased. Biased methods give particular agents priority due to the implementation. It is even more challenging to develop a method to also allow reproducibility such as replicating the sequence

of the serial model. Reproducibility in the context of ABMs is of importance, for example the issue of replication contributes towards the scepticism and perceived lack of robustness when using underdeveloped ABM applications in public health [160].

A list of notable GPU ABMs with conflicting issues are as follows:

- Sugarscape Model - Discrete space model of agents that move around to look for sugar. Conflicts occur when two agents attempt to move into one cell.
- Segregation Model - Discrete space model in which two type of agents which move if they are unhappy. Happiness is based on the number and type of other agents surrounding each agent. There is a conflict when two agents try to move into one cell.
- StupidBugs Model - Discrete space model that is based on the StupidModel where bugs/agents randomly move around with larger bugs pre-empting smaller bugs. Conflicts occur when larger bugs pre-empt smaller bugs for actions such as movement into a cell.
- HeatBugs - Discrete and continuous space model, Bugs/agents move if the surrounding temperature in their environment is too hot or cold. The bugs radiate heat and in the discrete space model they cannot move into a cell that already has another bug.

A common issue with these models is the conflicting movement problem. The movement problem is when agents simultaneously attempt to move into a neighbouring 4 or 8 connected cell and only one agent is allowed to be in that cell. This means that in typical discrete space models, each cell may have a maximum of 4 or 8 other agents that attempt to move into it. There are several common conflict resolution methods for these ABMs on the GPU. The methods are listed and described based on the movement problem they resolve, they are as follows:

- *Overwrite Method*: - This method allows threads to overwrite each other so that the agent of the last thread is allowed to move into a conflicting cell.
- *Iterative Push*: Write agent id to 'claim' an empty cell, write priority order to depth buffer to find highest priority agents. Unsuccessful agents will try to claim a different cell in the next iteration.
- *Non-iterative Push*: Variant of the iterative push method in which all conflicts are resolved in one iteration by allowing threads to take control of agents that have relinquished their priority over a cell to another agent.
- *Iterative Pull*: Each agent registers there intent for a particular neighbouring cell and then the cells scans its neighbours to find the agent with priority to move into the cell.

If the *overwrite method* described in [161] is used for conflict resolution in the Animat model, it can be biased towards the last thread as agents updated by this thread will always have priority. Using this method would mean the simulation is not reproducible on the GPU as the scheduling of threads are not managed by the user. The *iterative push* method described in [162] uses depth buffers

which can only be read from after synchronisation at the end of kernel calls. This means conflicts are resolved one at a time and can be costly for many iterations.

In the *non-iterative push* method described in [163], threads take control of other agents they preempt within the same kernel call and update them. This means that these threads will need to load the data of the agents which can be costly on the GPU and particularly with the case of the Animat model. Agents of the Animat model have many fields that may need to be loaded, such as health, priority number and other agents within their neighbourhood. This case may result in undesirable memory access patterns that can have a negative effect in the performance.

The *iterative pull* method described in [164] avoids atomic operations by allowing agents to register their intent for a particular resource. This resource then scans its neighbouring agents to work out which agent has priority. This method iteratively resolves conflicts for one resource at a time. This is not suitable for the Animat model as there is potentially thousands of prey in each cell that would need to be iteratively resolved. This method would require an excessive number of iterations to resolve the conflicts in the Animat model. Moreover, there could also be thousands of predators in the neighbourhood that would need to be iteratively scanned which could be costly on the GPU.

These methods can be effective when resolving conflicts in simple situations such as the movement problem as there is a maximum of 4 or 8 agents that can move into each cell for typical discrete space models. Movement is not conflicting for agents in the Animat model, many agents can freely move into each cell as more than one agent can occupy a cell. The behaviour of predators eating prey is conflicting as there could be many predators that simultaneously try to eat a prey. The problem for the Animat model is that within a Moore neighbourhood there could be thousands of predators and thousands of prey and each predator can only eat one prey. This means that these methods can be detrimentally slow for the situation of the Animat model when implemented on the GPU.

1.6.3 GPU ABM frameworks

There are some notable frameworks that assist the implementation of ABMs on the GPU. Although these simplify some of the implementation requirements such as providing pre-implemented storage, any form of conflict resolution still has to be implemented by the user. The storage may be optimised towards the communication protocols provided by these frameworks, however this can restrict which methods can be used to resolve conflicts. A list of the frameworks are as follows:

- TurtleKit - A Java Library using JCUDA for multi-agent systems (MAS) to perform partial computation on the GPU [165].
- MCMAS (many-core MAS) - High level Java interface that links with OpenCL to perform partial computation on the GPU [166].
- MASS (Multi-agent spatial simulation)- C++ API that links with CUDA and provides basic simulation environment for agent-based models [161].
- Flame GPU (Flexible large-scale agent modelling environment for the GPU) - a template based simulation environment that maps formal descriptions of agents into simulation code [167].

The TurtleKit and MCMAS frameworks simplify the process for the user to move some computation to the GPU then after to return the results back to the CPU to continue the simulation. However, memory transfers between the host and device are costly if performed each simulation cycle. The MASS API offers storage for agents on the GPU and a concept of places that represent the environment on which agents can interact with. The agent movement problem for MASS requires the user to implement a conflict resolution scheme. Each place/cell stores an array of four agent pointers for each direction that agents can move from into that cell. This array can then be accessed by the conflict resolution scheme. This approach may require an *iterative pull* resolution method and may be useful for the simple movement problem with a maximum of 4 or 8 agents per cell. However, this requires a lot of additional storage. In the case of the Animat model, movement is not a problem as there is no limit to the number of agents that can occupy a cell. If this system is used for conflict resolution for Animat agents it would require a lot of memory for the dynamic aspect of many agents per cell and would require the use of the unsuitable *iterative pull* conflict resolution method.

Flame GPU uses the XML schema to represent agents with their functions implemented with the C language. Agent communication is achieved through the use of messages that are stored in a globally accessible list. An example list of messages is the coordinates of all agents in the simulation. This framework does not provide an execution environment the agents can interact with. Flame GPU provides support for discrete and continuous space ABMs with the agents defined either as discrete or continuos types. There are backend optimisations for the typical discrete type agents but this means that discrete agent population sizes cannot change during execution and must be initialised with a size that is a power of two and must be squarely divisible. Creation of discrete type agents during run-time is not allowed and there is only support for creation of continuous agent types. These conditions mean that Flame GPU may not be suitable to implement the Animat model. Moreover, the concept of messages means that the user is limited to using a conflict resolution method such as the *iterative pull*. This is due to the user defining interaction through C functions that are parsed and do not have access to CUDA functions such as atomic operations.

Storage is an important aspect of implementing ABMs on the GPU. The storage should provide efficient access for threads to read or write from agent information as this may negatively impact the performance of the implementation on the GPU. The MASS API stores discrete space agents using the AgentPool data structure described in [159] which is designed to handle dynamic agent creation and deletion. This storage consists of an array of structures to store agent data. An additional array consisting of indices are used to access agents in the agent storage array. Deletion of agents result in free slots within the agent storage array and as the agent storage array is not compacted, a taken flag array is used to track these slots. This allows dynamic creation and deletion of discrete space agents which are not addressed in the Flame GPU framework.

In the Animat model the discretised cells of the environment can be occupied by many agents. This means that the number of agents that can move into a cell is dynamic. Animat agents only interact with other agents in their own cell or neighbouring cells, thus it is of importance to try and store the agents sequentially for efficient access on the GPU. The MASS AgentPool storage approach may not be suitable for the Animat model as the agents that are stored are not moved around in each cycle. This means neighbouring agents may be stored sequentially in the first step but as agents can

move, these sequentially stored agents may not be neighbours in the next step. This may get worse as the simulation progresses. The array of structure storage used by the MASS framework may not be suitable for the Animat model as particular actions such as movements may only need to consider fields such as the location of other agents. Using an array of structures means that the location field for each agent is not stored sequentially and may not result in efficient access on the GPU.

1.6.4 Summary

The conflict resolution methods and frameworks previously discussed, all aim to address the common issues that occur when parallelising ABMs on the GPU. Issues such as the common agent movement problem for discrete space ABMs can be addressed by several conflict resolution methods as discussed. These are suitable for the common cases when there is a fixed maximum number of agents that can move into a particular cell. In these ABMs the maximum number of agents for Von-Neuman neighbourhoods is 4 and for Moore neighbourhoods 8. However, for the dynamic case of the Animat model there are thousands of agents that may occupy and cause conflicts in a Moore neighbourhood. It is not uncommon to have thousands of predators and prey in a Moore neighbourhood. The predators may attempt to eat the prey simultaneously and only one predator is allowed to eat one prey. A conflict resolution method must address this dynamic case and provide improved performance on the GPU. The method also needs to support the aspect of reproducibility such as the replication of a serial model. This means conflicts may have to be resolved base on a priority sequence.

The ABM frameworks for the GPU can simplify the process of describing and storing agents for parallel update. However, as these frameworks target the simple movement problems in ABMs they do not address the uncommon parallelisation problems of the Animats model. These frameworks restrict the methods in which conflicts can be resolved and using an *iterative pull* method to resolve the dynamic conflict situations of the Animat model can result in poor performance on the GPU. The Animat model ABM has uncommon problems such as the varying number of agents that can occupy a cell. The agents would need to be stored in a way that encourages desirable memory access patterns on the GPU to result in efficient update of agents. This storage would also need to suit a conflict resolution method developed to support a parallelisation strategy to achieve improved performance for large-scale Animat simulations.

1.7 Research Questions

The Animat agent-based model extends the typical discrete space model topology by allowing many agents to occupy a cell in a lattice environment. This can lead to large population numbers of agents that can interact locally and result in emergent system behaviour. A typical Animat simulation with a few hundred thousand agents can be simulated on a single-core CPU in a feasible time-frame. There is little to no work simulating the Animat model at a large-scale typically with millions of agents. This is due to the limitations of the computational power of a single-core CPU used to compute an Animat simulation. The main focus of this thesis is to parallelise the Animat model on the GPU hardware to improve the performance of the simulation. This is a challenging task to achieve when an

improvement in the performance cannot sacrifice original model behaviour. The work in this thesis considers the following main question:

Can the Animat model be implemented on GPU architectures to improve performance without sacrificing original model behaviour?

The main question can be decomposed into specific technical sub-questions that may need to be considered together:

1. What parallelisation conflict resolution method is required to implement the Animat model on the GPU to increase the system size that can be feasibly simulated?
2. What data-structures support efficient storage and access to satisfy the programming model of parallel architectures?
3. Can the original model behaviour be reproduced using general purpose graphics processing units to simulate the Animat without sacrificing performance?
4. Can the agent behaviours of the parallel Animat simulation still be easily configured for experiments without requiring the user to have a sufficient level of CUDA programming skills?

1.8 Structure and Contributions

The original contributions presented in this thesis are as follows:

- Development of parallel methods for computing the Animat agent-based model on GPUs with improved performance and without sacrificing original model behaviour (see Chapter 5).
- Identified, and formulated a data-structure to support the dynamic storage requirements for the Animats model while providing efficient memory access patterns (see Chapter 5 particularly Section 5.4).
- An improved method for resolving conflicting and ordered agent actions in parallel (see Chapter 5 particularly Section 5.7).
- A GPU backend based on Lambda functions that simplifies the process of programming agents with varying behaviours (see Chapter 7).
- Investigation of optimisations using pre-processing for Animat agents (see Chapter 5 particularly Section 5.10).

This thesis should be read in the order of the presented linear sequence to understand the model, the issues, the solutions and the results that conclude the work. Chapter 1 provides an introduction to parallel computing and the field of Animat research. This chapter introduces the history of artificial-life research and the related macro- and microscopic predator-prey computation models.

Chapter 2 provides a complete description and discussion of the Animat model. Chapter 3 presents the implementation details of the original model on a single-core CPU. In Chapter 4 dependencies are identified to develop a multi-core implementation of the Animat model that consists of a domain decomposition approach to parallelise an update of the simulation. Chapter 5 presents a new update strategy to improve the performance of the Animat simulation using GPU architectures and a conflict resolution system to retain original model behaviour which is discussed in Chapter 2 and 3.

Performance results are provided in Chapter 6 to compare the parallel implementations of a typical Animat model using a range of system sizes. This chapter also discusses how the behaviour of the original model can be reproduced on GPU architectures and provides an analysis of the conflict resolution system.

Chapter 7 covers an internal, agent definition systems that improve ease of modification to individual behaviour. Chapter 8 provides an overview of the thesis, concludes and discusses the implications of this work. A discussion of some possible future work is also covered in this Chapter.

2
Animat Model

Agent-based Animat is an Artificial life model developed to explore and understand intelligence through computer-generated artificial life-forms [108]. Early computational ALife models such as Avida [168] and Tierra [169] helped to pioneer the Animat field [30, 32]. The Animat approach has been used to model life-like behaviour such as predation, herding and reproduction [53, 170–172] which is commonly referred to as emergent phenomena [173].

The composition of the model consists of multiple autonomous interacting Animat agents or individuals that exist in an artificial world. Animat agents typically have an internal state, behaviours and goals. The internal state may change to represent their situation and may also influence the behaviours to achieve their goals; behaviours can be defined as a set of rules (rule set) with each rule governing an action to achieve a purpose. Individuals may belong to a species or type by sharing common characteristics or attributes and have the capability to learn or adapt. An example is in an Animat model designed to explore the effects of a survival trait on the carrying capacity of the species population such as the predator's ability to use camouflage [115].

The interaction of agents typically involves other agents or the environment in which they are situated. Common environments may consist of various topologies such as networks, lattices or continuous 2-dimensional and 3-dimensional space. The implementation of the topologies may also vary; an example is the geometries such as triangle, square and hexagonal formations of a lattice. As typical interactions between agents are limited to neighbouring cells in a lattice environment, the lattice formation decides who and what they can interact with. An example of an agent and environment interaction is a herbivore type species grazing to consume resources from the environment.

The update of the model occurs in cycles, and each cycle consists of a randomised order of Animat agents that decide and perform an action that may influence the state of the next cycle. A typical cycle may include other requirements such as health and age toll, recording movement of Animat agents, adding new agents or removal of deceased individuals.

The emergent behaviours of the model are often analysed to understand the relationship with the micro-level rules of individuals that can be goal-driven, adaptive or simply reactive. The emergent phenomenon is an important aspect of agent-based Animat models, and sufficient descriptions in the context of ALife is provided by Chris Langton [1], John Holland [174] and Mark Bedau [175]. The experimentation with the Animat model has resulted in various emergent properties such as ag-

gregate patterns [109], defensive spirals [53], swarms [176], segregation [113] and spatial tribes [110]. Although Animat is intended as an agent-based predator-prey model in this dissertation, it can be used to represent other types of agents such as soldiers in the Battle of Isdandlawan [177]. Figure 2.1 shows examples of the spatial defensive spiral clusters that emerge at various update cycles of a typical Animat model simulation.

2.1 Animat Agents

Animats are autonomous, interacting predator-prey agents with the capability of acting without external direction when encountering any given situation [63]. The agents are composed of individual attributes that define their state and influences their behaviour. Behaviours are described as actions in Subsection 2.1.4 and individual attributes in Subsection 2.1.1. Individuals can belong to a group that have common attributes to define a species and have the ability to interact with other agents or their environment. Interaction and individual environments are described in Subsection 2.1.3 and attributes of a species is discussed in Subsection 2.1.2.

Simple Animat agents only require minimal individual attributes or behaviours such as a location to identify their position and movements towards a goal. A typical Animat agent extends a simple individual by including fundamental attributes and behaviours that borrow abstract ideas from animals. Typical Animat agents include parameters such as age, health and common behaviours consist of actions such as breeding, eating and grazing.

2.1.1 Individual Attributes

Individual attributes are combined to define the state of an Animat agent at any simulation timestep. The minimum requirements to be an Animat agent is the existence in a model with the ability to occupy or move to other locations in the environment. Thus the only attributes required are position and status to represent the state of the individual as either dead or alive and occupying a particular position in the environment. Additional attributes can be included to model various aspects such as an age attribute to limit the life span of an Animat agent.

An individuals mutator attribute is a variable that may change over the life span of an Animat agent to represent its current situation such as location or age. Animat agents may also include fixed attributes which are commonly used to describe or identify the individual. Example fixed attributes are gender and a type of identity such as a unique number.

A typical Animat agent may include mutators such as age, health, position and status. The age variable is a numeric value to gauge and limit the time of existence of the Animat agent. A whole number is used to represent age and can accumulate during each model update cycle, the concept of an update cycle is elaborated in Section 2.3.

The health attribute is the most decisive variable for a typical Animat agent as the value influences the behaviour of an individual. For example, if the Animat agent is rational in a sense, then it should prioritise the behaviours of searching and consuming food resources to replenish health before searching for a partner to reproduce.

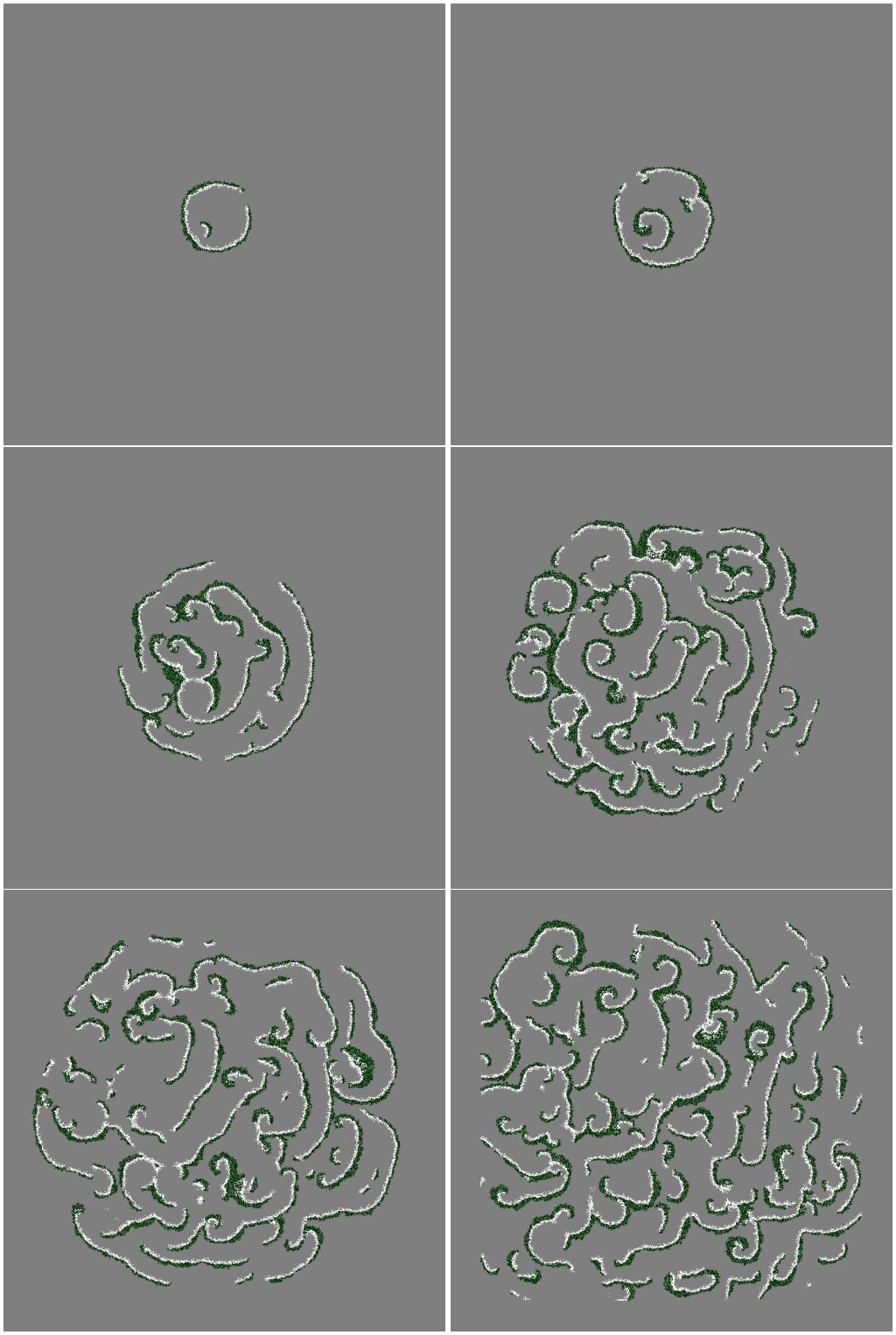

Figure 2.1: Simulation screen captures of the Animat simulation on a 600^2 system size at specific steps from left to right 200, 400, 600, 800, 1000 and 1200. This sequence of screen captures shows the emergence of spatial defensive spiral clusters at various update cycles.

Table 2.1: Typical attributes of an Animat agent in which the value *N* may represent different ranges for each attribute.

	Description	**Value**
Age	Model time of existence	0 - N
Health	Energy or health state	0 - N
Position	Location based on environment type	(x,y) , (x,y,z)
Gender	Gender Type	Male/Female
ID	Identification	0 - N
Symbol	Species Symbol	F (Fox)/ R (Rabbit)
Marker	General marker	A-Z or 0 - N
I Status	Status at current step	Alive/Dead

The position variable represents the location of the Animat agent based on its world or environment, and the model environment is described in Section 2.2. The status variable is used to represent the existence of the Animat agent and may influence the behaviour of other individuals. An example is that an Animat agent may move away from a particular direction if too many dead agents are identified in a specific area. The status value can extend to provide extra measurable properties such as the type of death e.g. eaten, old age or hunger. This additional information allows the model to record collective data for a species.

The fixed attributes serve to identify or categorise individuals. The ID attribute uniquely identifies the Animat agent while the species symbol classifies the group an individual may belong too. The marker can be used to group the agent by specific, measurable properties such as type, spatial groups or herds etc. Gender can be a requirement of the model and can be used to identify potential partners for breeding.

Table 2.1 shows the individual attributes of a typical Animat agent that consists of a combination of mutator and fixed type variables. Agent complexity may be defined with the addition or removal of optional attributes. An example is giving the Animat agents a camouflage ability by including a numerical value to represent a percentage chance that the individual is hidden [115].

2.1.2 Species Attributes

Species attributes can be represented as inheritance rules and constraints that are commonly shared to characterise a breed that individuals belong to. The inheritance rules can be used in the breeding process to regulate how attributes or behaviours are passed on to new Animat agents. Constraints may represent some species trait in which individuals must follow. Typical predator and prey species consist of a combination of inheritance rules and constraint attributes.

The constraints are maximum age, health, vision, birth-rate and behaviour influencing rules such as crowd breeding, crowd grazing and grazing success rates. Attributes such as maximum age, health, vision and birth rates can be defined to copy abstract traits from animals such that a predator may live longer than prey or have larger vision ranges as a part of their hunting abilities. It is possible for Animat agents to have varied definitions of these attributes; for example, individuals may use different maximum vision range or health values. In these cases, the attribute belongs to an individual. The constraints of a typical Animat model are described as follows:

Maximum age is a limit of the amount of model time an Animat agent is capable of surviving given optimal circumstances. Limiting the life-span of Animat agents may induce the natural concept of boom-bust cycles observed in predator-prey models to occur. These limits generally relate to the size of the model environment and portray an existence with spatially bounded means, i.e. migration from one farthest point to the other is not possible.

Maximum health is a limit in the Animat model that depicts an abstract limitation to energy or health values of Animat agents. Typically the individual's health attribute value is increased by behaviours such as consuming prey or grazing. The maximum health restriction also serves to represent the abstract idea of being filled with energy to prevent individuals from repeatedly performing specific actions such as eating, grazing or hunting.

Maximum vision is a value in which the area of an individuals vision may be derived from. The definition of the value depends on the model environment; for example, lattice types may use value to calculate a circular area of vision whereas network types could specify the number of nodes. The model in this thesis uses a lattice type environment, and the maximum vision value can be defined based on the model's environment size.

The birth-rate of a species is a value that models the abstract idea of circumstances an animal may encounter during the breeding process. The concept of a birth-rate simplifies the complex circumstances of breeding in animals. A higher success rate for the prey species could be interpreted as the preys ability to procreate an abundance of offspring. The predators lower success rate may be perceived as the lower number of offsprings produced. The value is typically defined as a percentage to represent the breeding success of a species.

Crowding constraints such as breed crowding and graze crowding serves to influence dispersion of Animat agents when a particular spatial area has become too dense with inhabitants. An integer value can be used to enforce these constraints. The value represents the maximum number of individuals allowed in a specific area before some behaviours such as breeding or grazing are prohibited.

The species inheritance rules such as clones, crossover, all standard, and mutation chance all apply to the inheritance part of the breeding process. Enabling clones means offsprings will inherit the same behaviours from the mothering Animat agent. Crossover implies there is a mix of obtaining a subset of behaviours from each parent. All standard implies all Animat agents of a species will exhibit the same behaviours. Mutation chance offers a percentage the behaviours could mutate in a particular way by including or removing new rules or altering the priority of the actions.

In the typical Animat model, the species attributes are shared as fixed values by default, and it is possible for it to be dynamic. The dynamic aspect may be used to evolve a species throughout the simulation. An example species specification of predator and prey species is shown in Table 3.1.

Table 2.2: Value representation for commonly shared attributes of the predator and prey species with N.A specifying an attribute not applicable to that species.

	Predator	**Prey**
Clones	True/False	True/False
Crossover	True/False	True/False
All Standard	True/False	True/False
Mutation Chance	True/False	True/False
Maximum Age	>1	>1
Maximum Health	>0	>0
Maximum Vision	>1	>1
Birth Rate	0 - 100	0 - 100
Breed Crowding	>= 0	>= 0
Graze Crowding	N.A	>= 0
Grazing Success	N.A	1 - 100

2.1.3 Individual Environment

The environment of Individuals can be described as the area in which they are able to see or interact. This localised area of vision is defined by the attribute of an individual or species. In a typical Animat model, the circular vision area is set by the radius variable which is a member of the species attribute; however, it is acceptable for a model configuration to specify varying sizes of vision area for individuals.

The individual's environment typically contains other Animat agents or environmental resources such as availability of grass with a visualisation is shown in Figure 2.2. Although interactions are limited to direct neighbours, It is possible to extend the model to expand the term localised by sharing information between Animat agents in herds with specific markers. This provides the abstract idea of community-oriented behaviours such as sentinels acting as a communal threat alert.

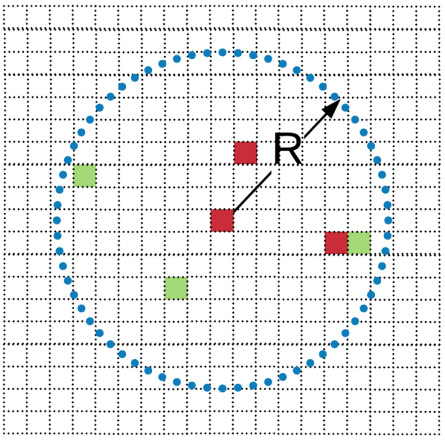

Figure 2.2: Individual environment of an agent located in the center that shows other agents within a vision area defined by the radius R.

2.1.4 Actions

Animat agents are autonomous and self-directed as they function independently when making decisions to interact with other Animat agents or their environment. Interactions between individuals are guided by their behaviours that consists of actions that can be performed based on their current situation. Typically individual behaviours are specified based on the species they belong to, for example, prey generally grazes, and predators usually hunt and consume prey.

A rule consists of an action and condition and in order for an action to be performed the condition must be met. For example, the action of eating is usually performed when an individual is hungry and the condition to represent hunger is that the current health level must be below a specific percentage. The description of the typical rules are as follows:

Seek a Mate: To seek a mate the Animat agent observes the area within its vision range to find the closest agent of the same species to move towards. If gender is an aspect in the model, the agent must locate the other gender of the same species. This action requires the condition that the individual is not hungry with a health level greater than fifty per cent. Definition of condition values is based on model requirements. Varying the value would adjust the significances of maintaining health.

Move in a Random Direction: The Animat agent attempts to move randomly with a fifty per cent chance the action may fail as a condition. This action is active but non-reactive to the environment or other Animat agents.

Breeding: Breeding requires a partner to be in the same or adjacent cell, and there is also a requirement that the health level is greater than fifty per cent. The condition for the health level to be above fifty per cent indicates breeding to be equally as important as hunting and eating.

Predator Seek Prey: Predator specific rule that requires the health level to be below fifty per cent. The health level percentage indicates when the predator recognises it is hungry and will attempt to look for prey to hunt.

Consume Prey: Predator specific rule that also requires a prey to be in the same cell or adjacent cell to be eaten. The health level percentage must be below fifty per cent to indicate hunger.

Graze Grass: Prey specific action where individuals will attempt to graze grass from the environment for nutrients. The health level must be below fifty per cent to indicate hunger.

Move Away: This action can be used by the Animat agent to move away from its own or other species. Generally, the prey would apply this rule to either flee from predators or move away from other prey due to overcrowding. There are no health requirements to perform this action, but the avoided species must be in the same or adjacent cell.

Table 2.3: An example rule set of a predator and prey agent.

	Predator Condition:	**Prey Condition:**
Seek Mate	Health >50%	Health >50%
Seek Prey	Health <50%	N.A
Consume Prey	Health <50% and Prey Adjacent	N.A
Graze Grass	N.A	Health <50%
Breed	Health >50% and Mate Adjacent	Health >50% and Mate Adjacent
Evade Predators	N.A (Apex Predators)	Predator Adjacent
Move Away From Own Species	Crowded Neighbourhood	Crowded Neighbourhood
Randomly Move	50% Chance	50% Chance

The simplest form of an Animat agent only requires basic movement rules while an agent of a typical predator-prey model may need a set of rules to abstract species behaviours. For example, predators may have rules to represent an ability to hunt and eat prey and prey with rules to graze or evade predators.

The rules expressed in Table 2.3 refer to typical actions required to be an independent functioning Animat agent of the predator and prey species. Each rule is simple in itself; however, when combined they can provide complex emerging behaviours in spatial systems [53]. The pairing of conditions is a fundamental requirement to provide some rationality to the actions that can be performed.

The descending rules in Table 2.4 represent the priority of the actions that form a rule set for a two species predator and prey Animat model. The rule names remain unchanged for consistency as they are the terms used in publications by the original Animat authors. Similar to the conditional pairing in Table 2.3 some rationality is applied to the priority of rules that factor in determining the behaviours of Animat agents. Typically a rule set is specified based on a species, however, the priority of the rule set may vary for individuals.

Table 2.4: Default rules priority for a typical Animat simulation.

Predator Rule Priority	**Prey Rule Priority**
Seek Mate	Breed
Seek Prey	Graze Grass
Consume Prey	Seek Mate
Breed	Evade Predators
Randomly Move	Move Randomly

2.2 Model Environment

The model environment is the world in which Animat agents exist, within the environment agents are able to move, communicate or interact. Earlier models considered the lattice topology on a two-dimensional plane that included cell shapes such as squares, triangle and hexagons. These geometries were explored due to the Animat model borrowing a property that cells neatly share common boundaries from cellular automator models [178]. The analysis from the different geometries identified the differences in movements are due to the connectivity of the varying cell types. Square

lattices used a Von Neumann neighbourhood template, while triangular has three and hexagonal six connecting cells. The cell connections are visualised in Figure 2.3.

Figure 2.3: Geometric lattice neighbourhood connections from left to right: square, triangular and hexagonal.

Modern typical environments consist of a two-dimensional lattice of square cells, and each cell represents a space where there is no limit to the number of occupying Animat agents. The lattice grid map typically starts from a size of 200 by 200 cells and can expand infinitely. Smaller environment sizes of a typical Animat model may result in a population collapse.

The modern implementation of the Animat model uses an artificial boundary where cells outside of the boundary do not contain grass. Animat agents may wander out passed these boundaries but can quickly die of starvation. The artificial bound provides a population natural boom-bust mechanism whereas strict boundaries where Animat agents cannot move to may not allow this as prey on strict boundaries can easily change directions and remain alive. It is possible to extend the lattice to periodic boundaries which connect the bordering cells to create a torus-shaped environment which may result in interesting emergent behaviour.

2.3 Update Cycle

An update cycle is a concept that each step of a model represents a cycle of existence where Animat agents decide and perform an action to reach a goal. The update of a typical Animat model is used to describe an update cycle in this section. An update cycle consists of synchronised processes such as randomising the order of agents for an update, the decision and perform action stage and the state update of agents as a result of their actions.

In a typical model, the processes must follow some rules, the rules are discussed, and the processes are described as follows:

Randomising the Animat agent update order is an essential process as this removes the artefacts that may occur as a result of bias arrangements, such as a sweeping effect based on the spatial locality [178]. All individual should have the same chance to decide and perform their actions first. This process should occur at the start of an update cycle.

The stage at which Animat agents decide and perform actions begin after a random order is determined. This stage involves an individual incurring a health and age toll for the cycle before they can decide to perform any actions. If the toll renders the individual dead, i.e. health is zero or age has reached the maximum limit, then the Animat agent is marked dead for removal which occurs at the end of the update cycle. If the Animat agent remains alive after a health and age toll

update, it will decide which actions to perform based on its current situation and the priority of its rule set.

Movement actions may change the position of an Animat agent which is recorded at the end of the cycle. During this stage interactions may result in the change of status of other individuals such as predators eating prey setting their status from alive to dead and dead prey cannot be consumed by other predators. Another outcome of interaction is the creation of a new agent as a result of the breeding action, new agents are typically added to the model at the end of the update cycle, and crucially only one action is permitted per agent in an update cycle.

The last process of a cycle is the part where all Animat agent states are updated as a result of the actions they have performed. The movements are recorded, the dead agents removed and new agents are added. As previously mentioned this is an update cycle for a typical Animat model in this thesis, the processes of other models can be formulated differently to represent an update cycle.

2.4 Emergent Behaviour

An interesting feature of the Animat model is the emergent behaviour of the system as a whole. The appearance of complex phenomena occurs when the state of the model has progressed to a point where noticeable patterns emerge from random or directed collective behaviours of Animat agents. Collective behaviours include a group of predators hunting a group of prey.

Movement actions such as manoeuvring towards or away from individuals are used to create simple rules to abstract animal ideas such as searching for partners, food or avoidance. Combining simple movement rules in a prioritised order may collectively produce segregation of species in their environments. Moreover, when propagation behaviours such as breeding are included in the rule set, emerging aggregate patterns appear and can be categorised as waves, spiral curves, blobs or transitional clusters [53].

The environment size is also a factor for spatial emergent patterns. Typically, lattices with dimensions of 200^2 or more units are required to provide enough spatial locality to allow defensive spiral patterns to emerge which is seen in the left of Figure 2.4. Large transitional clusters observed on the right of Figure 2.4 emerge as a result of Animat agents using strategic searching patterns on an example 600^2 system size [179].

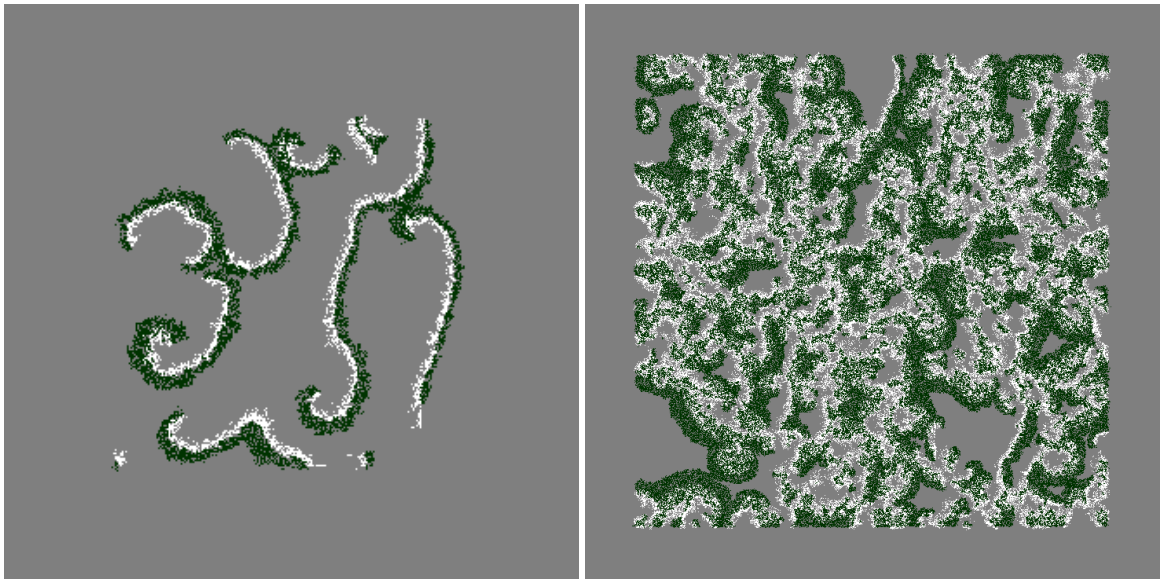

Figure 2.4: Visualisation of simulation state at 5000 steps for standard and strategic searching patterns from left to right.

2.5 Summary

The properties of an Animat model such as the Model rules, Animat agents, species behaviour, environment, individual Animat behaviour, agent communication and interaction have been described. Typical species actions and rational conditions have been identified with a description of a lattice model environment in which Animat agents exist. The update cycle of a model is discussed and a solution to implement a computational simulation of a typical Animat model is presented in Chapter 3.

3

CPU Implementation of the Animat Model

The design of a framework and simulation software to study ALife began in 2004, the fundamental aspects of an Animat model construct, include the model laws, roles of communication, geometrical environments and framework technologies [108]. Analysis for the Animat software revealed sufficient complexity to consider low-level languages such as C or C++. The Java language was used in some implementations of the model and a virtual Prolog approach to impart a degree of reasoning to simple agents was explored [180]. This chapter discusses the requirements of the modern framework in terms of a typical predator-prey Animat model described in Chapter 2.

To support a typical predator-prey Animat model the framework is required to represent the agents and their environment and allow the simulation to be progressed by rules for the update cycles of the model. To represent the agents, storage is required for individual and species attributes while functional support is needed for behaviours that occur in their individual environment. The model environment requires storage to represent the lattice cells that include resources such as grass. The simulation update must support the update cycle requirements of the model such as randomising the order of agents for an update, updating agents decisions, and recording the new state of individuals for the next cycle as a result of their actions.

A visualisation of the connections between the components that make up a typical Animat model is shown in Figure 3.1. Animat individuals are connected to the rules, species data, environment and their storage. These connections represent potential interactions such as the Animat agent reading information from their environment to influence their behaviours. The environment may store information about Animat individuals which require access to the Animat storage.

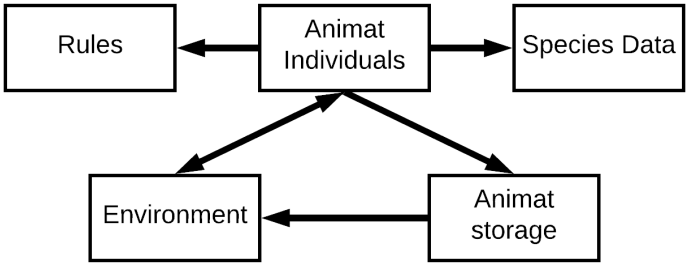

Figure 3.1: Connections between storage and functional requirements of a typical Animat model.

3.1 Animat Agent

Animat agents are represented as an instance of the Agent class with a set of data members that store the state of the individual at a certain time-step. This is implemented using C++ classes and encapsulation to protect the primitive type variables or custom data structures. The actions are public methods that have access to protected mutators and attributes. The blueprint of an Animat agent can be seen in Listing 3.1, where the type *Agent* pointer **next* is a protected pointer to the next *Agent* in the same cell location of the environment. The functionality of the environment discussed in Section 3.4 describes the connection between the requirements of an Animat individual and the Environment. The *rule_set* instance of a *RuleSet* class functions as a storage for an agents rule set. The *RuleSet* class is discussed in Section 3.3. This can be useful in specific model configurations that require individuals to have permutations of the rule set; however, another approach exists for the typical case which is described in Section 3.3. Actions and rule sets are discussed in Chapter 2.

Listing 3.1: Animat Agent Class

```
1  class Agent{
2  protected:
3     // Agent attributes
4     Agent *next;
5     int x, y, new_x, newy, age, health;
6     char symbol, gender, usedrule;
7     int id, marker, mother_id, species_index;
8     int n_rules;
9     std::string load_behaviours;
10    StatusType status;
11    RuleSet rule_set ;
12
13 public:
14    Agent(){}
15    Agent(int a_id, int a_species_index, int a_age, int a_health, int a_x, int a_y, int
          a_new_x, int a_new_y, std::string a_load_behaviours);
16    ... // Other Constructors
17
18    Agent(const Agent& other);
19    Agent(Agent&& other);
20    Agent& operator=(const Agent &other);
21    Agent& operator=(Agent &&other);
22
23    bool move_away(SpeciesType target_species_type);
```

```
24    bool move_towards(SpeciesType target_species_type);
25    bool breed(XORWOW *rn_generator);
26    ... // Other Agent Behaviours
27
28    virtual ~Agent(){}
29 };
```

3.1.1 Requirements of Agent Actions

Actions of Animat agents that are discussed in Chapter 2 can be represented as C++ functions or methods. Action functions are implemented with operations to represent the processes of an action and may result in transactions such as reading or writing to the attributes of an individual and the environment or creation of new *Agent* instances that are added to the Animat storage. The functionality may be required in all actions or just one. A common requirement is searching and interacting in the Moore neighbourhood while an example unique requirement is adding new agents when breeding. The implementation requirements of a typical Animat agent action is listed as follows:

- Search and interaction with agents in the Moore neighbourhood.
- Search of agents within a vision area.
- Adding new agents.
- Removing agents.
- Environmental storage access for resources such as grass information.

The search and interaction between Animat agents within a Moore neighbourhood is a frequent process that Animat agents are likely to perform each step. Animat agents search for other agents by accessing the storage of environment grid blocks. The grid blocks contain *agent* pointers that are discussed in Section 3.4. The access to the environment grid blocks can be done using search offsets. Search offsets are coordinates that are stored in a list which is sorted by the distance from a central location. Search offsets are discussed in Section 3.7.

Searching of Animat agents within a vision area or an individuals environment is similar to the Moore neighbourhood search. However, the searching begins two cells away, and the move towards action is an example rule that requires a vision area search. The vision area search for either a partner or food source requires the search offsets of all cells within the Animat agents vision radius. The offsets are ordered closest to farthest in a circular pattern for the Animat agents to iterate through cells to find the nearest other agents.

Depending on the model requirement, interactions may only be possible for Animat agents one cell apart. Animat agents may have different searching strategies that are based on factors such as distance and densities of target individuals in certain directions [179]. The data structures that can be used to store Animat agents can have an effect on the cost of search operations, these data structures are discussed in Section 3.5.

The random aspect is implemented using pseudorandom numbers to evaluate the chance to attempt specific rules. Two actions in a typical configuration requiring the random element are breeding and grazing. These actions use a generated random number to compare with species attributes such as breeding success or grazing success to determine if the related action can be performed. This can extend to custom rules with the requirements set in the species data storage.

Newborn Animat agents can be added within the same update cycle when the breeding action is performed or they can be added for the next update cycle. If they are to be added in the next update cycle then the newborn agents can be stored in a temporary list. This list is appended to the main storage at the end of the current cycle.

The breeding function that requires this functionality is also required to specify the state of the new agent to be born. The attributes of the new agent can be clones of parents, have mutated rules, a share of parents rules or some form of an inheritance rule. The exact behaviour is defined by the configuration of the model.

The functionality to remove agents is only required at the end of each time-step, this is also known as phase two of the update which is discussed in Section 3.6. Although predation kills prey, the prey is only marked with an eaten status which flags for their removal during phase two. Depending on the data structures used, any trace of the dead agent must be removed from all storage such as the link-list of agent pointers in each cell of the environment discussed in Section 3.4.

A typical Animat model has grass nutrients in each cell location of the model environment which can be accessed by Animat agents for consumption by means of modifying the environment attribute. The grass nutrient can be fixed allowing for unlimited resources or dynamic to represent limited food that is replenished at specific rates.

3.2 Species Data

The Species information that is shared between individuals of a group is stored as a C structure shown in Listing 3.2. An array of *SpeciesData[n]* is used where n is the number of species. The majority of variables are common between species apart from *crowd_grazing* and *grazing_success*, which are restriction variables that belongs to species that graze. Variables that are not relevant to the species can be set to zero during the initialisation phase.

Listing 3.2: Species Structure

```
struct SpeciesData {
   int index;           // Index number for this species
   char symbol;         // Letter symbol
   int total;           // Number of agents for this species
   int main_id;         // Used to allocate id numbers
   bool clones;         // True if each child is a clone of its mother
   bool cross_over;     // True if each child uses crossover on its parents' rules
   bool all_standard;   // All animats use standard rules - ignore clones, crossover, etc
   int mutation;        // Percentage chance of rules mutating (zero = no mutation)
   int crowd_grazing;   // Crowding can prevent grazing
   int crowd_breeding;  // Crowding can prevent breeding
   int grazing_success; // Chance of successful grazing
   int birth_rate;      // Percentage chance of giving birth
   int search_cut_off;   // Maximum number of search off sets within vision range
   int max_vision, max_health, max_age;
};
```

Table 3.1: Categorisation and value representation for data of a species in which the value may represent different ranges for each type of species data.

Species Data	Type	Value
index	descriptor (`int`)	0 - N
symbol	descriptor (`char`)	F (Fox)/R (Rabbit)
total	count (`int`)	0 - N
main id	count (`int`)	0 - N
clones	birth (`bool`)	True/False
cross-over	birth (`bool`)	True/False
all-standard	birth (`bool`)	True/False
mutation	birth (`int`)	0 - 100
crowd-grazing	restriction (`int`)	>= 0
crowd-breeding	restriction (`int`)	>= 0
birth-rate	restriction (`int`)	0 - 100
search cut-off	restriction (`int`)	0 - N
max-vision	restriction (`int`)	>1
max-health	restriction (`int`)	>0
max-age	restriction (`int`)	>1

There are various types of species data which serve different purposes, and these can be categorised as descriptor types, count types, birth types and restriction types. The categorisation of species data and value representations are shown in Table 3.1. Each categorised type can be represented as an `int`, `char` or a `bool`.

The descriptor types are *index* and *symbol*. The *index* is the location of the data set within an array and *symbol* is a character that is used to identify the species. The count types are *total* and *main_id*, *total* is used to track the total number of live agents for each species and *main_id* is used to assign unique identification to each Animat agent. The range of data types is considered when the *main_id* is used to assign unique identification for every Animat agent that has existed in a simulation run as that number could exceed the range of the utilised data type.

The birth types such as *clones*, *cross_over*, *all_standard* and *mutation* functions to guide the initialisation of behaviour and state of new Animat agents.

When *cross_over* is enabled the *mutation* value represents a percentage that the cross over mutates and depending on the model configuration, it is possible to allow mutations that only come from the mother. *Clones* represent a copied rule set from either of the parents and a *all standard* configuration assigns a fixed rule set to all new Animat agents.

The restriction parameter types such as *crowd_grazing*, *grazing_success*, *birth_rate*, *search_cut_off*, *max_vision*, *max_health* and *max_age* represent restrictions to the behaviour and actions of each species. If limitless grass for a grazing species is enabled, defining a restriction to the number of Agents in a Moore neighbourhood encourages spatial movement and discourages dense population cluster growth. The parameters *grazing_success* and *birth_rate* add a random element to the success of the grazing and breeding behaviours of an Animat agent. The parameters *search_cut_off* and *max_vision* limits the inter-actability and vision range of Animat agents. The parameters *max_health* and *max_age* caps the energy and existence time of an Animat agent.

Depending on the configuration of the Animat model, the species data can change dynamically.

For example in a dynamic environment where the grass is a limited but replenish-able resource, the crowd grazing restriction may be lifted as spatially limited food supplies naturally encourage migration of Animat agents.

3.2.1 Species Class

C++ inheritance is used to implement the Animat agents to simplify expansion to multiple species. The base class consists of the fundamental attributes and behaviours or actions that are relevant to all species. The fundamental attributes include *Id*, *Age*, *Symbol*, *Status*, *Coordinates* and *Health*. The operations include movements, eating and breeding. The derived classes are prey and predator, and the latter can extend to an apex predator species. The derived Animat species may have different behaviours for fundamental actions which require different implementations; for example, prey grazes grass and predators eat prey. Breeding may require separate implementations as gestation periods, or the number of offspring can vary. Figure 3.2 illustrates an Apex predator class that is derived from the predator class by inheriting fundamental attributes from the Animat class and selective operations from the predator class.

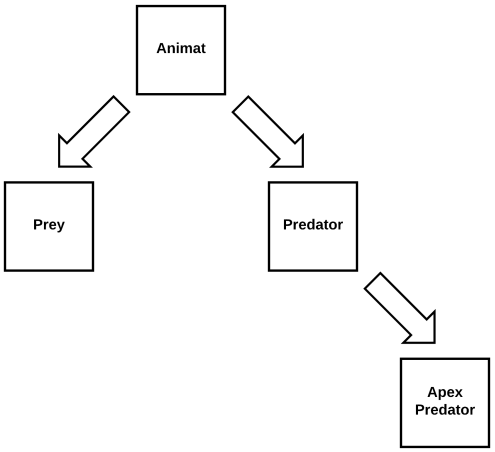

Figure 3.2: Animat agent class hierarchy with agents as the base class that can be derived to species such as prey, predator and apex predator.

3.3 Rules

Each Animat agent has a set of rules that define their behaviour. The rules are discussed in Chapter 2 and are executed in order of priority until an action can be successfully performed. The rule set is implemented as a C++ class object seen in Listing 3.3.

This class stores information for all the rules that make up a rule set for an Animat agent. The `rules_priority string` represents the priority of the rules of the rule set. For example an arbitrary `rules_priority string` set to *rgbam* has 5 characters *'r'*, *'g'*, *'b'*, *'a'*, *'m'*. Each one of these characters is the code for a rule with the priority of the rule sequenced from left to right.

Table 3.2: Default rule requirements and specifications of a typical Animat simulation where well-fed and hunger level is represented by a number from 0-9 and the target species and sex are F/R for Fox or Rabbit and M/F for male or female.

Rule	Well-Fed	Hunger Level	Target Species	Target Sex
Move Towards Mate	5	0	F/R	M/F
Move Towards Prey	0	5	R	N.A
Eat Prey	0	5	R	N.A
Breed	5	0	F/R	M/F
Graze Grass	0	5	N.A	N.A
Move Away	0	0	F/R	N.A
Move Randomly	0	0	N.A	N.A

Each rule has a condition that must be satisfied before the action of the rule can be performed. Each action may contain extra information that applies to it. For example, some actions may only apply to a specific gender or species. Table 3.3 is a list of rules with their requirements and specifications. These rules may only apply to a particular species such as only prey agents can graze. These rules can be combined to create a rule set. The information for all rules of a rule set represented as a *RuleSet* instance is stored within the private `char*` members of the *RuleSet* class.

In a typical Animat implementation an example prey agent's `rules_priority string` can be *rgbam*. The first character *r* represents the code for the move away from predators rule. The second *g* is for the graze rule, *b* is for the breed rule, *a* is for the move away from other prey rule and *m* is for the move towards other prey rule.

An example composition of the move away rule can be *AXR30* where character *A* is the action name, *X* means no target gender to apply the action on and *R* is the prey species to apply the action on. The last two characters *3* and *0* are the hunger level and well-fed conditions to attempt the action. The decimal value of the hunger level and well-fed `Char` can be used to calculate a percentage. The value 48 is deducted from the decimal ascii value to convert a `char` to and `int`. For example the decimal value of *3* is 51, the formula is $(x - 48) * 10$ so the hunger level percentage is 30 percent as $(51-48)*10 = 30$. The well-fed level being *0* means there is no well-fed condition as $(48-48)*10 = 0$. These percentages are used by the Animat agent to check if they satisfy the conditions for that action. The check is simply done by comparing the agents current health to the condition percentages. The health value must be below a hunger percentage, and above the well-fed percentage, this effectively allows the Animat agent to iterate through actions in the *RuleSet* in priority order and decide which to perform based on its current state.

The rule information for each action is set by the user and certain configurations may produce simulation wide complex phenomena such as spatial cluster configurations discussed in the publication [113]. The `Char` type for indexing used in the Animat implementation are arbitrary and can be represented using other data-types such as integers, for example 0 can be move-away, 1 for move-towards, 2 for breeding etc. This implementation uses the `char` data-type as a byte for storage is sufficient to represent the variable.

Listing 3.3: RuleSet Class

```
class RuleSet {
private:
   int num_rules; // Number of rules that belong to this rule set instance

   check_condition *check_conditions_array; // Array of check function pointers
   do_action *exec_action_array; // Array of action function pointers
   std::string rules_priority; // String of rule priority e.g "rgbmR"

   char *code;       // Used only to recognise the rule
   char *action;     // A - away, B - breed,
   char *gender;     // F/M - only use this rule if gender matches
   char *species;    // A species symbol, eg: foxes eat (R)abbits
   char *hunger;     // EXR[3]0 - eat if health LESS THAN 30% (3 = 30)
   char *well_fed;   // BF00[5] - breed if wellfed (5 = 50%)

public:
   RuleSet(std::string rule_str) : num_rules(rule_str.size()), rules_priority(rule_str)
       {...};
   std::string get_rule();
   void set_rule_info(int index, char code_char, std::string rule);
   void create_rule();
   void set_func_pointers(do_action *d_a_arr, check_condition *c_c_arr);
   void use_action_func_ptrs(Agent *agent);
   ~RuleSet();
};
```

The anonymous function pointers seen in Listing 3.3 allows the implementation to avoid switch case statements of up to N number of action cases for comparison to each corresponding action code. Referencing the action function pointer calls the function at the pointer address and effectively avoids worst cases of N number Animat agents times an N number action code comparisons per time step. An example of 80,000 prey with 6 elements in the rule set and 50,000 predators with 5 elements in the rule set would equate to a worst case $(80000 * 6) + (50000 * 5) = 730,000$ character comparisons each time step.

3.3.1 Rule Set Implementation

There are two options for the implementation of rule sets: fixed or dynamic rule sets, as previously mentioned a fixed rule set means that each Animat agent of each species will share the same prioritised rule set; thus it can be termed a species rule set. A dynamic rule set means that each Animat agent may have a different priority of rules within a rule set.

There are two categories for a dynamic rule set, the first is referred to as a birth rule, this means the conditions of a rule and its priority within a rule set are defined at birth or creation time and cannot change afterwards. The second is an evolving rule set which allows the conditions of a rule in the rule set to change over a range of update cycles. For example, prey that move away from predators in adjacent cells may start to move away from predators that are two cells away.

An array of *RuleSet* instances named a ($ruleSetArray$) can be used to store fixed and dynamic rule sets. The *RuleSet* instance is an object of the RuleSet class shown in Listing 3.3. The *RuleSet* class is described in Section 3.3. A $ruleSetArray$ for a fixed rule set may have only one *RuleSet* instance while a dynamic rule set may have many. The number of *RuleSet* instances in a $ruleSetArray$ for a dynamic rule set is determined by the number of possible permutations of a particular fixed rule set,

for example, a fixed rule set with 6 rules will give a factorial of 720 possible unique combinations for a dynamic rule set. To use the $ruleSetArray$ for dynamic rule sets, the $ruleSetArray$ can be populated with permutations of *RuleSet* instances. They can be sorted in lexical order so that a method can be used to search the index of particular *RuleSet* instances within the $ruleSetArray$.

An index to a random permutation of a *RuleSet* can be allocated to agents at birth. The agent can use the index to access their rule set. The index can be located using the function in Listing 3.4. This function searches for the index to the *RuleSet* instance which is stored in a lexically sorted order within the $ruleSetArray$. This function compares each `char` rule from the *RuleSet* instance and finds the index of where the *RuleSet* instance is stored in the $ruleSetArray$. This is achieved by calculating offsets that can be formulated base on the number of rules in the *RuleSet* instance and the total permutations of the *RuleSet* instance.

To simplify the description of this process, `int` numbers *0 - 9* are used in the place of the `char` types of the *rules_priority* `string` from a *RuleSet* instance. This may help to visualise the lexical ordering. For this case an arbitrary *rules_priority 10534* is used in the place of the previous example: *rgbam*. The values *'1'*, *'0'*, *'5'*, *'3'*, *'4'* has no relation to each of *'r'*, *'g'*, *'b'*, *'a'*, *'m'*.

The *rules_priority 10534* has 120 unique permutations. Example permutations of *10534* can be *50134*, *01534* or *40531*. If the permutations are lexically sorted and stored in an array then all the permutations are grouped in the sequence of which all lexically sorted permutations starting with *0xxxx* are stored first. Permutations starting with *1xxxx* is second, *3xxxx* third, *4xxxx* fourth and *5xxxx* fifth. The variable *x* represents a value from the original *rules_priority 10534* which is *0, 1, 3, 4* or *5*.

These can be referred to as subdivided sets or subsets with the first-level of subsets containing 24 permutations each, as the total 120 permutations divided by the number of rules which is 5. Each of the first-level subsets can be further subdivided to contain the second-level subsets. For example the first-level subset starting with *0xxxx* contains 4 second-level subsets sorted in the sequence *01xxx*, *03xxx*, *04xxx* and *05xxx*. The size of each level of subsets can be calculated by dividing the product of the factorial, which for this case are: $120/5 = 24$, $24/4 = 6$, $6/3 = 2$ and $2/2 = 1$.

Another way to store dynamic rule sets are to dynamically add or remove permutations of the rules. This can be effective for cases where the number of permutations of a rule set exceeds the expected average number of Animat agents per step. For example in Table 3.3 A *rules_priority* string of size 10 has 3,628,800 permutations. All Animat agents that have existed in a simulation run may not assign all the possible permutations. As the permutation of *RuleSet* instances increase, a C++11 *std::unordered_map* implemented using hash tables can be used for storage due to the expected complexity of the required operations, $O(1)$ for insertion, deletion and look up.

Individual storage is required for cases with large numbers of permutations to sensibly store in an external container. An instance of a *RuleSet* for each Animat agent can be stored as private members of the Agent class. An example of a possible configuration that requires individual storage is a situation where Animat agents evolve not only their *rules_priority* string but also the rule condition for each action. An Animat agent that only eats when their hunger level is below 30 per cent may evolve that value to a higher number, this may be due to environmental scenarios where food sources are scarce thus prioritising eating over other actions may benefit survival.

Listing 3.4: Set Rule Index

```
void set_rule_index(Agent *agent, std::string rule, std::vector<std::string> *
    permutation_array, int *range_size_array, int total_permutations){

   int rule_index = -1;
   int num_rules = rule.size();
   int current_range_start = 0;
   int current_range_end = total_permutations;

   for (int rsi = 0; rsi < num_rules; rsi++){
      int curr_range_size = range_size_array[rsi];
      char curr_char = rule[rsi];
      int skip = 0;
      for(int i = current_range_start; i < current_range_end; i += curr_range_size){
         std::string current_string_checked = permutation_array->at(i);
         if(curr_char == current_string_checked[rsi]){
            current_range_start = current_range_start + (skip * curr_range_size);
            current_range_end = current_range_start + curr_range_size;
            skip = 0;
            break;
         }else{
            skip++;
         }
      }
   }
   rule_index = current_range_start;
   agent->ruleset_index = current_range_start;
}
```

A function is used to set the function pointers to the check condition and corresponding action. Listing 3.5 is an example of the function which can either exists as a member function to the Animat agent or externally to handle both individual *RuleSet* instances and external *RuleSet* containers. The function sets the check condition and corresponding action function pointer by using the relevant rule information retrieved for the action code. The rule information such as hunger level and the well-fed level is passed to the condition check whereas rule targets such as gender or species are used in the action functions. The use of Lambda wrappers is to maintain the same type and number of parameters which call action functions with a varying number of parameter and types. This preserves the readability and ease for maintenance of action functions.

Table 3.3: Number of permutations for each number of rules: 1 - 11.

Number of Rules	Number of Permutations
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800

Listing 3.5: Set Agent Function Pointers

```
void Prey::set_agent_actions(std::string rule_str, int rule_index){
   // Set rule information
   this->Rules[rule_index].action      = rule_str[0];
   this->Rules[rule_index].gender      = rule_str[1];
   ... // Other rule information
   // Set function pointer for rule condition
   this->Rules[rule_index].check_hunger_fed = [](int i, Agent *agent){
      return static_cast<Prey*>(agent)->action_hunger_fed_precheck(i, SpeciesInfo[1].
          max_health);
   };

   char action_code = this->Rules[rule_index].action;
   // Set function pointer for rule action
   if(action_code == 'm'){
      this->Rules[rule_index].do_action = [](Block **Grid_Blocks, Agent *agent, KISS_prng *
          rng){
         return static_cast<Prey*>(agent)->move_towards(Grid_Blocks, SpeciesType::prey);
      };
   }
   else if(action_code == 'R'){
      this->Rules[rule_index].do_action = [](Block **Grid_Blocks, Agent *agent, KISS_prng *
          rng){
         return static_cast<Prey*>(agent)->move_randomly(rng);
      };
   }
   // Other actions
   else if(...){
      ...
   }
```

The perform rule process is now simplified to a single loop instead of switch case statements. Listing 3.6 is an example of individual *RuleSet* instances being executed by calling the action function pointers only if the corresponding check condition function pointer is satisfied and returns a true statement. The action function also satisfies the Animat update method by returning a true statement if the action can be performed or else the next pair of condition and action is tested. There are possible scenarios whereby an Animat agent may not succeed in any actions. For an external *RuleSet* container, the function pointers do not require modification as the agent that needs the rule can simply be passed in as a parameter.

Listing 3.6: Execute Agent Rules

```
for(int r = 0; r < agent->get_n_rules(); ++r){
   if(agent->get_rule(r).check_action_condition(r, agent)){
      if(agent->get_rule(r).do_action(Blocks, agent, &RNG[rng_index])){
         break;
      }
   }
}
```

3.4 Environment

The environment for predator and prey in the modern Animat model consists of a two dimensional spatially square geometrical lattice grid. The neighbouring connection for this implementation uses

a Moore neighbourhood; a past study has also implemented a 4-connected or Von Neumann neighbourhood [181]. Each cell in the grid represents a discrete X and Y area and interaction between Animat agents are typically limited to its cell and the surrounding neighbouring cells. There is no maximum number of agents that can occupy the same cell; however, there can be conditions that restrict actions if there are too many agents in the same location.

Early implementations of the Animat model consisted of unbounded plains ensuring movement of Animat agents or agent clusters to be free of artificial constraints. Unbounded models tend to have the problem of growing exponentially to large population sizes. Artificial boundaries can be introduced to maintain constant population levels. The grid constraints can be circular or rectangular borders that prevent Animat agents from moving out of bounds. It was found that the localised properties of the simulation ensured artificial boundaries had no observable effect on the formation of Animat agent clusters or behaviour. Experimentations with the model can be done without boundaries affecting emergent properties [182]. Figure 3.3 is an example 8 by 8 cell lattice with a 4 by 4 celled grass area coloured in green where the prey species can graze for nutrients. It is possible for prey to move to non-grass areas coloured in grey as a result of behaviours such as predator avoidance; however, these agents tend to die as a result of hunger.

0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	2	1	0	0	0
0	0	2	4	3	2	1	0
0	1	1	3	2	5	0	0
0	0	0	1	4	3	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 3.3: Example 8 by 8 cell lattice and 4 by 4 green coloured grass area in the center of the environment with the number of occupying agents in each grid cell.

In early implementations of the Animat model searching required iteration through agents sorted by X and Y attributes of the environment in the storage container. Using only a sorted storage still does not provide quick indexing to relevant information such as the indices to the Moore neighbouring cells. Test runs of the Animat model with the default configurations show on average 2-3 Animat agents occupying a cell. This means there are likely multiple Animat agents in most locations so the use of a sorted environment storage of individuals does not provide direct access to the first Animat agent at a particular location. A data storage with indexing like a Compressed Sparse Row

(CSR) can be used to provide precise indexing without additional calculations. CSR is described in Section 3.5.

Another approach called the Hybrid combined [183] was introduced to improve performance. This solution consists of a constant two-dimensional array based on the height and width of the environment. Each element represents a cell in this array which comprises of a singly-linked-list of Animat agents. This approach improved complexity to $O(1)$ to access linked-lists at each cell location in the environment. However, this approach requires the additional memory of grid width times the grid height times the size of agent pointers and may not be suitable for other architectures.

The data structure to store grid cell information is implemented as a simple C struct called a grid square in the modern Animat implementation. A grid square contains environmental properties local to the cell such as fixed or dynamic grass value that can be stored as an integer. Depending on the model requirements, the grass value may be fixed at a specific amount and never decreases to portray an unlimited food source for prey. Other examples could be fixed crowding limits that are defined spatially to represent different environmental types at specific areas of the grid. The linked-list of Animat agent pointers is initially set to *NULL* in the grid box or grid cell and points to agents when populated.

3.5 Data Structures

An analysis of the storage requirements is necessary to determine suitable data structures to implement the Animat model. Generic data structures are available from the C++ libraries or concepts can be developed to implement a customised data structure to suit the needs of a typical Animat model. Every data structure has its target problem to address with their pros and cons that may not be suitable for all situations.

3.5.1 Storage Requirements

The storage requirements can be identified from the update process of Animat agents as energy costs, age costs and actions all require storage and access. At each time-step, the cost of energy or health and ageing rate may exceed or fall below limits that change the status of an Animat agent to dead. The eating behaviour of predators also render prey individuals decease, and dead Animat agents require the removal or reuse of their memory locations in storage which can be done by freeing the Animat agent object memory and resetting their associated pointers.

A majority of actions require interactions with neighbouring target individuals or a lookup of specific search areas. Interactions that predate other species such as the *eat* function has the ability to mark the predated Animat agent as eaten in turn requiring the same storage operations as death by other means. A primary factor of propagation, the breed function requires new Animat agents to be inserted into the storage container. Another propagation behaviour is movement, the movement action is either in a targeted, evasive or random direction and may require multiple operations to update the pointers of Animat agents in the storage. Operations are implementation dependent, and there are two used in this Animat implementation. The other solution where spatial mapping is tied

to the environment is described in the Environment Section 3.4.

The common storage operations required and identified for the Animat model is as follows:

- Insertion of Animat Agents.
- Removal of Animat Agents.
- Searching of Animat Agents.
- Movement of Animat Agents.

Profiling was done on the previous version of the Animat implementation, and it shows that based on the default configuration, on average there are fluctuations of 5000 - 20,000 insertion and 5000 - 20,0000 removal of Animat agents per step that have either been born or deceased in a typical configuration. The average number of live Animat agents per cycle in a typical configuration is 150,000 thus a worst case of 12-13 per cent fluctuation each step in terms of storage operations is required. The default actions of an Animat agent all require at minimum a search for the nearest target agent in a Moore neighbourhood, with the move towards action requiring a search of all cells within the vision radius. If the storage operations required for a typical Animat simulation are prioritised based on their average computational cost per step then the rank of these storage operations would be as follows:

1. Searching of Animat Agents.
2. Insertion of Animat Agents.
3. Removal of Animat Agents.
4. Movement of Animat Agents.

3.5.2 Storage Data Structures

Common data structures seen in Table 3.4 shows either a dynamic array/stand template library vector (STL) Vector or a binary search tree STL Multi-set is sufficient and suitable to store the Animat agents. A Vector is simple for use and testing of implementation concepts by providing an added advantage of $O(1)$ complexity for access. Although search is $O(n)$ on average and worst case, the Animat agents Vector can be maintained in a sorted by search key order. This essentially allows binary search algorithms to be implemented which turns search into an $O(log(n))$ complexity, providing better scaling for larger model sizes. However, the insertion and removal of Animat agent objects still pose a performance problem.

The spatial positioning of Animat agents in a lattice environment has similarities to a sparse matrix in which a nonzero element is represented as an occupied cell. A matrix can be defined as sparse if the total nonzero elements is below 50% of the total number of elements. A storage system that is based of the Compressed Sparse Row(CSR) format can be used to store Animat agents in a contiguous arrangement in memory and provide indexing for search. The CSR based storage represents a matrix by three one-dimensional arrays that are commonly referred to as A, IA and JA.

Table 3.4: Common Data Structure Operations

Data Structure	Time Complexity							
	Average				Worst			
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

A contains the nonzero elements and IA splits the array A into rows and JA aligns the values in columns. The CSR based data structure for Animat is implemented as a C++ template class seen in Listing 3.7. The vector A will store the Animat agents in sorted coordinate order and IA stores the number of agents on each of the Y dimension with JA storing the X dimension index of the agent.

Listing 3.7: Compressed Sparse Row Class

```
template <class T> class Csr{
   private:
   std::vector<T> A;
   std::vector<int> IA;
   std::vector<int> JA;

   public:
   Csr(int a , const int ia , int ja) : A(a) , IA(ia + 1 , a) , JA(ja, 0) {}
   void update();
   void insertA();
   void deleteA();
   ...
   ~Csr();
};
```

The benefit of a spatial indexing system minimises the number of operations for searching Animat agents as the indices in IA provides direct access to the first agent in each Y dimension while the JA provides information for the X. The other benefit of using the CSR format for the Animat implementation is the access of contiguous memory by the cache. Searching for Animat agents is user-defined and can be implemented using various strategies. The patterns either look for the nearest Animat agent or searches for all individuals in vision. For the nearest target search, the CSR based structure matches performance when compared to the Storage and indexing using the environments lattice of Animat linked-lists. Performance is improved when search patterns that require access to all cells in vision make use of the contiguous memory locations used for storage by the CSR based data structure.

The other option using a binary tree multi-set for Animat agents storage provides an average case complexity of $O(log(n))$ and a worst case of $O(n)$ for search, insertion and removal operations.

Multi-sets would scale suitably with experiments that have a large population number expectancy, and another benefit is duplicate keys of two-dimensional cartesian coordinates that can be used for ordering. The Moore neighbourhood and vision range searches on average cases affect the performance of the Animat implementation the most; thus a multi-set can be combined with the environments array of linked-list optimisation described in Section 3.7.

3.6 Multi Phase Update

An important aspect of modelling spatial agents is to specify how the individuals are updated or evolved in the simulation. The spatial agents are updated in either a fixed or random order where it might perform some action such as eat, breed, move or kill in turn changing the state of individuals to either grow, move or die. The update mechanism is typically an iterative process to evolve the phase or state space of the whole model [184, 185].

Before the original Animat simulation was developed various implementations of another model, the Eden model [186] were analysed. The Eden model was chosen due to simplicity and similarities to the Animat grid environment. The analysis revealed the sweeping problems of a fixed order sequential update resulting in skewed results. This was solved by randomising the order of sites for update to produce non-biased simulation data.

The two-phase system however still revealed some issues for the original Animat implementation. There was a resource consumption issue, and an example is when two predators would consume the same prey in a time-step. This issue causes an ambiguous model as it is not possible for one prey to sustain multiple predators. An example sequence of two predators A and B consuming Prey X:

- Phase One (Current State)
  - Predator A executes 'Eat' Rule on Prey X.
  - Predator B executes 'Eat' Rule on Prey X.
- Phase Two (Future State)
  - Prey updated by setting health to zero (dead status).
  - Predator A updated by increasing health from consuming Prey X.
  - Predator B updated by increasing health from consuming Prey X.

The resource consumption scenario has lower probabilities of occurring in small systems with $agents < 1000$; however, a small system size may prevent complex spatial behaviours from emerging [109] or cause the populations to collapse. The problem was revealed in the rendering of the simulation state at certain time-steps and the data plot of population averages. A solution to the problem was developed; however, it was not possible to retain a pure two-phase update system. The solution is a hybrid of the sequential and two-phase update system. This is possible by giving prey or resources extra attributes to record their state at each time-step, and the resource consumption scenario now executes as follows:

- Phase One (Current State)
  - Predator A executes 'Eat' rule on Prey X and sets Prey X state to dead.
  - Predator B attempts to execute 'Eat' rule Prey X but fails due to Prey X being dead.
  - Predator B attempts low priority rules.
- Phase Two (Future State)
  - Prey updated by setting health to zero (dead status).
  - Predator A updated by increasing health from consuming Prey X.
  - Predator B updated by modifying mutators based on another rule executed.

The addition of state variables for prey that can be modified means the system is not a true two-phase system. Sequential update is acceptable in these circumstances, although it is necessary also to randomise the order of agents for updating [187].

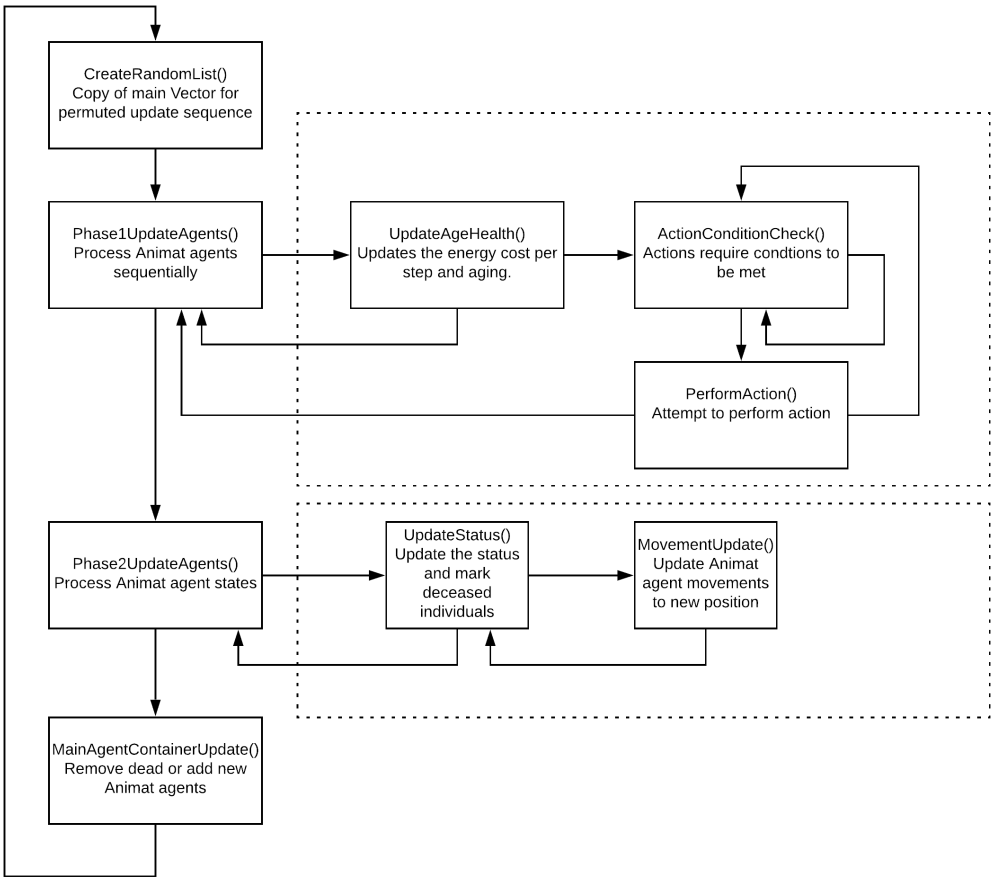

Figure 3.4: An illustration of the two phase update were phase one updates the agent and phase two updates the containers as a result of agent actions.

The simulation loop seen in Figure 3.4 describes the execution order of functions to perform a two-phase update. Phase one updates each Animat agent in a random order based on the current state. Phase two updates the state of Animat agents for the next step and this includes adding or removing individuals from storage containers.

3.6.1 Phase One

To avoid biased update orders that can cause sweeping problems, simple shuffling mechanisms can be used to randomise the order of agent updates. Two approaches are used in the modern sequential versions of the implementation. The first approach is a function called CreateRandomList(), written specifically to shuffle Animat agent object pointers. This function creates a copy of the main vector which is a relatively low overhead instruction. A loop till half the size of the copied vector performs swaps of random indices of pointers in the vector. The algorithm is essentially a modified version of the Fisher-Yates or Knuth Shuffle [188] and is seen in Listing 3.8. This algorithm is sufficient in avoiding the sweeping problem but may not provide random permutations.

Listing 3.8: Create Random List

```
void createRandomList() {
   Animal *swap;  int max_index, count, a, b;
   RandomList = AList;
   max_index = RandomList.size() - 1;
   count = max_index / 2;
   for(int i = 0; i < count; i++) {
      a = randomInt(0, max_index);
      b = randomInt(0, max_index);
      swap = RandomList[a];
      RandomList[a] = RandomList[b];
      RandomList[b] = swap;
   }
}
```

The second approach used the C++17 STL shuffle function. There are various implementations which have been deprecated or removed due to issues such as rand() being used. The problems include non-uniform distributions being used or full periods of the pseudorandom number generator being too low such as rand() on some Windows operating systems providing only 32767 states. The STL function called shuffle takes three parameters, the random Iterator first, random Iterator last and an STL random generator of type *UniformRandomBitGenerator* such as std::mt_19937. The STL shuffle function is the Fisher-Yates shuffle where the container to be shuffled is iterated in reverse order and swaps elements from a random index with the current element [189].

3.6.2 Phase Two

Phase two updates the state of the Animat agents as a result of their actions or step costs by changing their status, moving their pointers to linked-lists in new cell positions or removing the pointers from the current cells linked-list and Animat agent storage containers.

The first step involves erasing dead Animat agents from the main container by deleting the pointer to the agent, and this is crucially done after the agent pointer has been removed from the grid cell of the environment. New Animat agents are added to the main container and associated

grid cells by allocating memory before insertion into the linked-list in environment cells and the Animat agent storage. Health is updated for Animat agents that are still alive, and if the agent has moved, they are required to be removed from their last grid cell and inserted into the new grid cell in the environment. The process can be seen in Listing 3.9.

Listing 3.9: Phase Two Update

```
bool Animat::updatePhaseTwo() {
   switch(status) {
      case Ok : break;
      case OldAge : case Starved : case Eaten :
         GridRemove(this, x, y);
         return false;  // Agent has died
   }
   if((x != new_x) && (y != new_y)) {
      GridRemove(this, x, y);  // Agent leaves grid cell
      x = new_x;  y = new_y;     // Agent moves - but must be kept within the grid
      if (x < min_X) { x = min_X; } else if (x > max_X) { x = max_X; }
      if (y < min_Y) { y = min_Y; } else if (y > max_Y) { y = max_Y; }
      GridAdd(this);           // Agent joins grid cell
   }
   return true;
}
```

3.7 Optimisations

The profiling tools such as ValGrind and KCacheGrind can be utilised to analyse the Animat program performance. The default configuration was used to determine optimisation opportunities of the implementation with the searching behaviour of Animat agents deemed to be the costly process of Animat agent updates in phase 1. The nature of proximity as a variable required distance calculation of observable cells that contained Animat agents. As previously mentioned all but one of the default actions need a search of neighbouring cells. A simple but effective solution is to use a look-up table which is termed Search Offset Table in the Animat implementation. A further optimisation opportunity exists in pre-processing of neighbouring data that can be shared between Animat agents of the same cell. The tally of neighbouring Animat agents in a Moore neighbourhood is an example of information that can be shared between agents.

3.7.1 Search Offset Table

The initial Predator-prey Animat model constructed was implemented on an infinite plane with no spatially bounded constraints. The implementation used a list data structure to contain all the Animat agents with no spatial information that is directly indexable. Any searches of nearest neighbours quickly revealed a problem with $O(n^2)$ complexity in the form of a nested `for` loop to N in which N is the size of the container. Simple spatial partitioning was implemented in the form of an expandable grid whereby grid squares cover several locations and maintain a list of Animat agents that occupy cells in the grid square. The grid squares are generally the size $(larges\_vision\_range * 2 + 1)$, and new grid squares are created for Animat agents that wander out of the current grid square. The

expandable grid also posed problems as Animat agents tend to cluster requiring exhaustive $O(N)$ searches for nearest neighbours [190].

An alternate approach was implemented using a bounded matrix-system, a two-dimensional array or matrix where each cell is an open-ended list of occupying Animat agents [190]. The search for nearest neighbours takes place solely within a vision range thus improving execution speeds. The search is implemented using a nested for loop of adjusted start and end parameters based on the vision radius. The implementation inefficiency of distance calculations is still required to check if the cell in vision is within a circular area. A disadvantage of bounded plains is new emergent behaviour may be affected by the boundary constraints.

The current implementations extend the matrix-system by using non-grass locations to artificially bound the environment without compromising possible emergence [191]. The vision searches are also improved by using a search offset lookup table, and the result of this approach avoids repeated distance from centre calculations. The table consists of an array of search entry instances which contain the *X, Y, Offset* and *Distance* variables. The search entries are sorted by distance nearest to farthest of origin 0,0 (X, Y) with the corresponding *X* and *Y* offset which is used to calculate the index of grid cells from nearest to farthest to the Animat agent. Examples can be seen in Figures 3.5 and 3.6 where the radial lookup table prescribes the offset to the spatial cells that might contain another Animat agent.

```
( 0,  0)
(-1,  0) ( 0, -1) ( 0,  1) ( 1, 0)
(-1, -1) (-1,  1) ( 1, -1) ( 1, 1)
(-2,  0) ( 0, -2) ( 0,  2) ( 2, 0)
(-2, -1) (-2,  1) (-1, -2) (-1, 2)
( 1, -2) ( 1,  2) ( 2, -1) ( 2, 1)
(-2, -2) (-2,  2) ( 2, -2) ( 2, 2)
(-3,  0) ( 0, -3) ( 0,  3) ( 3, 0)
```

Figure 3.5: Vision radius of three (x, y) offsets.

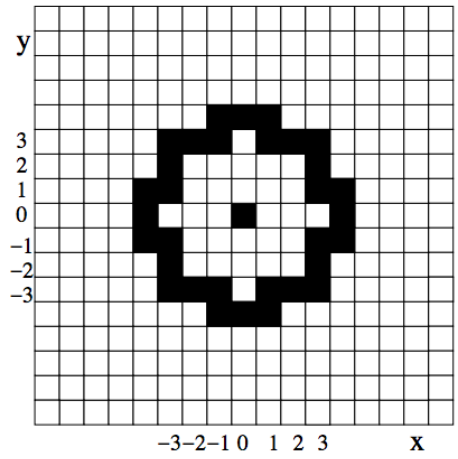

Figure 3.6: Border of cells within a vision radius of three.

The process of creating search entries involve an initial calculation and storage of all cell coordinate pairs within a circular vision range with the entries sorted nearest to farthest. A search cut off for each species can be identified by iterating the sorted search entries and comparing the distance of each entry to the max vision radius. Once the distance to the search entry is greater than the vision radius, the index can be recorded for the related species for each Animat agent to know how far to search.

3.7.2 Pre-processing Data

All but one of the core animat actions require information regarding their Moore neighbourhood. Move away requires another Animat agent nearby to move away from. Move towards requires no target agents nearby. Breeding and eating require Animats agents nearby to breed with or predate. Grazing and breeding require a count of the neighbourhood as prey cannot graze, and all species cannot breed if the area is overcrowded. The only action that does not require any neighbouring information is to move in a random direction.

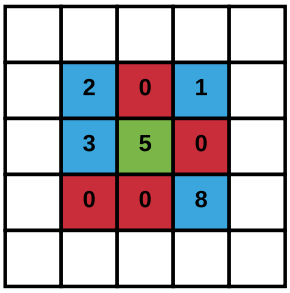

Figure 3.7: Moore Neighbourhood of the 5 agents occupying the green cell and various occupied neighbouring cells in blue with red containing no agents.

An example state of multiple occupying agents is shown in Figure 3.7 where if the 5 Animat agents in the green cell are required to search for neighbouring information such as the number of occupied neighbours then this would need 5 repeated searches and tallying of the same information.

To make use of pre-processing the environment grid cell structures can include two variables that keep count of the neighbouring predators and prey. To pre-process this information a two-pass system can be used with the first pass iterating through all grid cells to tally the total number of Animat agents for each species within a cell.

The second pass iterates through the grid cells and checks if there are any Animat agents occupying the cell which signals for a search through the neighbouring cells to tally the totals from the first pass for storage as the count of neighbours at the current cell. An example can be seen in Figure 3.8 with occupied green cells required to total the number of adjacent occupied cells.

All Animats agents attempting their actions can reuse the pre-processed neighbouring information. This can potentially save processing time if there is one or more Animat agent in a cell

attempting a rule that requires neighbouring counts. An Animat agent trying the move towards action can now skip to the next prioritised action if the neighbouring target count is greater than zero as there is at least one other agent within interaction distance. The actions breeding and eating can quickly be skipped if neighbouring target counts are zero which means there is no partner to breed with or prey to eat. Grazing is passed when the count exceeds the crowding limit.

Pre-processing can improve performance with model configurations that cause dense population clusters however not all pre-processed data may be reused as factors such as the chance to breed or graze can stop the related action that requires the neighbouring count to fail. In such cases pre-processing the data is additional operations that are not necessary. Pre-processing can be beneficial on the parallel architectures such as GPUs as there will be more efficient and fewer memory transactions.

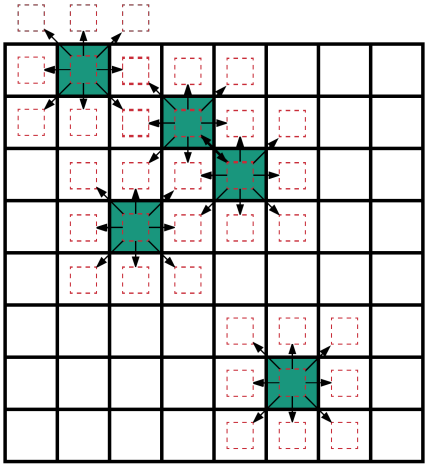

Figure 3.8: Green cells indicate one or more occupied Animat agents that may require a tally of other occupying agents in the surrounding Moore neighbourhood in red dashed squares.

3.8 Summary

The Animat model is composed of the environment, species data, Animat agents and their rules. The requirements are described and the connections identified with data structures and update strategies provided to implement an easily modifiable simulation of a typical Animat model on the C++ programming language. Performance optimisations such as search offset tables and data pre-processing have been presented which may help to experiment with large populations of the Animat model on a CPU. A parallel processing approach for multi-core architectures is provided in Chapter 4 to increase computational power that may be required for experimentation with larger model sizes of a typical Animat model.

4

Multi-core Implementation of the Animat Model

An implementation of a typical Animat model proposed in Chapter 3 can be extended on to a multi-core architecture to improve computational performance to explore larger than typical model sizes of 600 by 600 environments. The increase in model population size can be a result of increasing environment sizes or introducing variations to the model configuration that affect population numbers such as a change in the breeding rate of a species. These changes to the model may produce simulations that cannot be analysed in real-time and may take an infeasible amount of time for data collection. The multi-core developed is used as a means to analyse dependencies of the model when multi-threaded parallelism is introduced.

The use of multi-core architectures widely available on the modern personal computing devices can improve the performance of the Animat model simulation by utilising all of the available cores to compute the simulation. The multi-core implementation for this project runs on architectures with two or more cores. Due to the model requirements and dependencies, the expected performance increase relies on the parallel implementation strategies to achieve a speedup of close to N available cores. The number of cores or speed up required for real-time visualisation and analysis largely depends on the complexity of Animat agent behaviours. Larger than typical model sizes that result in increased populations may require a lot of memory transactions or dependent operations that can increase computational complexity.

There are a few notable application programming interface (API), framework, platform or packages such as CILK Plus, POSIX Threads (pthreads), Open Multi-Processing (OpenMP), Message Passing Interface (MPI), or Threading Building Block (TBB) to make use of the parallelisable hardware. These API can achieve parallelism by targeting specific problems or solve similar problems in different ways. Each API uses different levels of language, such as Pthreads using low-level and OpenMP, high level. The description and a brief comparison of the paradigms to implement the Animat model on the multi-core is described in Chapter 4.3.

The parallelism strategy for the Animat multi-core implementation is based on the analysis of the phase updates. The Animat model revolves around the Animat agents behaviour, and interesting,

complex phenomena arise from the outcome or configuration of the agent's behaviour over many simulation update cycles. The phase updates previously described in Chapter 3 process the decision making of the Animat agents, perform the actions and update the simulation storage containers. The phases can be analysed to reveal computationally intensive parts and identify implementation dependencies.

On a single core implementation, Animat agents are updated sequentially one at a time, and on multiple cores, multiple agents are expected to be updated simultaneously, which may cause problems such as resource contention or race conditions. The implementation approaches for a multi-core system is to either map the threads to Animat agents or the grid cells of an environment. The multi-core implementation in this thesis explores the approach of Spatial Domain Decomposition(SDD) where threads are assigned to spatially partitioned portions of the environment in an attempt to decrease parts that may require serialised processing such as addressing race conditions. The Spatial Domain Decomposition(SDD) approach is discussed in Section 4.2. The implementation of this parallelism strategy is discussed in this chapter by developing the data structure that is based on the dependency analysis of the Animat model described in Section 4.1.

4.1 Dependency Analysis

Dependency analysis of the Animat model is required to develop solutions to parallelise the simulation on multi-core architectures. The serial implementation discussed in Chapter 3 can be used to identify serial portions or stages of the model that can be parallelised so the overall performance can be improved when computationally intensive portions are effectively parallelised.

Analysis of data or functional dependencies can be established when investigating stages of serial implementation such as initialisation, ordering and phase updates with a visualisation shown in Figure 4.1. The description of the stages are summarised as follows:

- Initialisation includes initialising the agents and data containers required for the simulation.
- Ordering prepares the Animat agent pointers for non-biased updates by shuffling the pointers for a permutation of the update order.
- Multi-phase update processes the Animat agent behaviour and updates the data-structures as a result of agent actions that may require new agents to be added or dead agents to be removed.
- Visualisation and data output is not a requirement of a model update cycle and may execute every N number of steps.

The synchronisation is required between each stage as one must complete before the other can begin execution. The dependencies and synchronisation requirements of each of the stages is discussed in the subsections that follow.

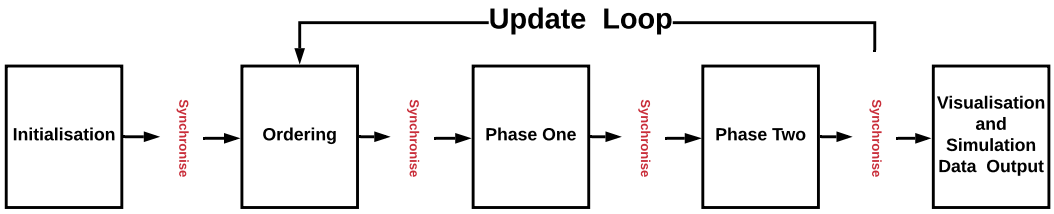

Figure 4.1: Synchronised stages of a serial implementation of a typical Animat model.

4.1.1 Initialisation and Ordering

The initialisation stage of the program involves loading configurations, initialising random number generators, loading the state of Animat agents and calculating the lookup table of search offsets. The ordering section includes randomising order of Animat agents for update.

The functions that are executed in the initialisation stage serves to configure the simulation or initial requirements of the model such as random number generators and can be named `readConfigFile()`, `readMapFile()`,`readRuleList()`,`setupSearchTable()`,`readAnimatStates()` and `initRng()`. A description of each of the functions is as follows:

- `readConfigFile()` - Reads from a file to load the species data.
- `readMapFile()` - Configures the environment.
- `readRuleList()` - Specifies the requirements or conditions of the rules.
- `setupSearchTable()` - Computes a lookup table of cell offsets for searching by Animat agents.
- `readAnimatStates()` - Loads the state of Animat agents saved to file from another simulation run.
- `initRng()` - Initialises the random number generator for the simulation.

All the initialisation functions access different memory locations and containers; thus there are no data dependencies and is only required to be executed once at the start of the simulation. Ordering involves random permutations of the update order of Animat agents and is required at the start of each update cycle. The initialisation and ordering stages do not take a noticeable amount of time compared to the update of Animat agents and the required maintenance of storage in a serial implementation of a typical Animat model.

4.1.2 Phase Updates

The dependency analysis of the phase one update revolves around the requirements of the Animat agents behaviour. The assumption each agent is updated sequentially no longer holds, and certain

behaviours may require some form of execution order management as they may be acting or performing operations that could cause race conditions. An example is if two predators were updated at the same time and both attempt to eat the same prey. They may both read the status of the prey as available and eat the prey to add nutrients to their own health. This would be incorrect as in a typical Animat model only one predator can eat one prey. The following are rules of a typical Animat model which can be categorised by actions that are independent and dependent.

Independent Actions:

- **Move Away:**
  - Requires access to Moore neighbourhood cells.
  - Modifies own *new_x* and *new_y* coordinates for the next state.
- **Move Towards:**
  - Requires access to all cells in range of vision.
  - Modifies own *new_x* and *new_y* coordinates for the next state.
  - Move Towards - Changes own *new_x* and *new_y*.
- **Move in Random direction:**
  - Modifies own *new_x* and *new_y* coordinates for the next state.

Dependent Actions:

- **Breed:**
  - Requires access to Moore neighbourhood cells.
  - Modifies the main container or a temporary container by adding new agents to them.
  - Inserts agent object into own grid list which is shared by other occupying Animat agents.
- **Eat:**
  - Requires access to Moore neighbourhood cells.
  - Modifies own health variables and status of prey consumed.
  - Modifies status of an agent that has been eaten and modifies own health values.
- **Graze:**
  - Requires access to Moore neighbourhood cells .
  - Modifies own health values.

The three rules identified to be potential causes of race conditions are eating, breeding and grazing. The eating rule can cause race conditions in situations where two or more predators that are updated simultaneously attempt to consume the same prey.

Dependent on the model configurations, breeding may cause race conditions in the Animat storage. In the case of new Animat agents being included in the same time-step, allocating memory and inserting new agent object pointers into associated storage may cause conflicts. Interactions with new agents could be ambiguous as there is no guarantee a neighbouring thread interacts with the new agent before or after the inclusion process. Adding new Animat agents at the end of the time-step does not pose a risk of conflicts given that the memory allocation process is thread safe.

Grazing can also cause race conditions, If grazing allows prey to update their health values based on the nutrients consumed in the same time step, then update order priority should be considered when interacting with that prey. In the situation where a predator attempts to eat the prey that is grazing simultaneously with a higher update priority, there is no guarantee that the prey completes the grazing action first and therefore providing the predator with the nutrient of the prey inclusive of grazing.

Figure 4.2 is an example spatial configuration where the numbered cells represent predators and the letters for prey. In the case where all predators are hungry and searching for prey to eat, then a sequential update of predators could process the actions without any conflicts. However, if all four predators are to be updated simultaneously this poses a problem as prey B is a possible food source for predators 0, 2 and 3 and prey D for predators 1, 2 and 3. Prey A can only be eaten by predator 0 and prey C by predator 3, and no predators are within range to consume prey E.

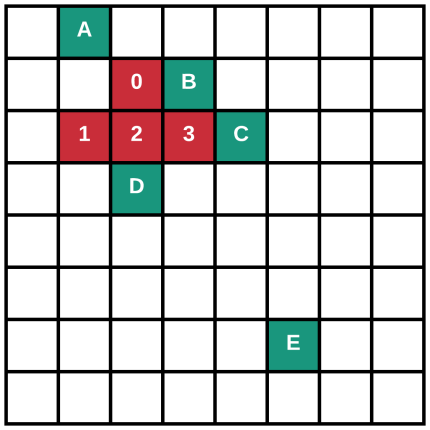

Figure 4.2: Simulation state with predators in red cells that can consume prey in green cells.

Phase two requires the synchronisation of all threads updating phase one to complete before execution. Phase two maintains the data containers for the next time-step, and depending on the implementation, could cause conflicts with memory allocated to Animat storage or environment data structures. The storage for Animat agents the environment are discussed in Chapter 3. Model configurations where Animat agents that are born and included in the same time-step could cause

issues if two or more threads attempt to insert new agents into the storage simultaneously.

As each environment lattice cell can sustain multiple Animat agents, issues may occur if the storage for the environment is not thread-safe. Figure 4.3 is an example spatial configuration where Animat agents represented as letters indicate their intent to move to another cell shown with red arrows. If all Animat agents in Figure 4.3 were to be processed simultaneously then agents B, C, D and E would require some form of thread management to so safely move them to the grey cells. No other agents are trying to move into the grey cell that agent A is moving to so there are no issues to process agent A simultaneously.

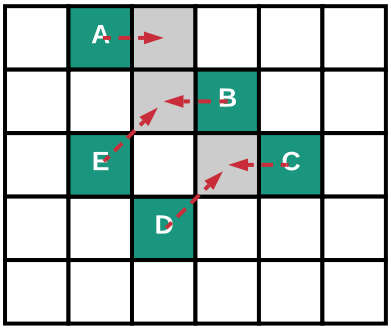

Figure 4.3: Agents coloured in green intending to move to cell locations coloured grey.

4.2 Parallel Method

This section describes the Spatial Domain Decomposition(SDD) of the environment to support the multi-core simulation design to update the Animat model on multiple CPU cores. This includes the description of how the environment is decomposed into spatial blocks and the use of C++ Classes to implement SDD.

4.2.1 Spatial Domain Decomposition

The spatial domain decomposition approach is designed to partition the environment into blocks or tasks that can be assigned to threads of a CPU core for processing. Past research has implemented update strategies to assign work by rows but does not address the potential problems of multi-threaded agent communications [192]. The SDD implementation adjusts block sizes based on the sparseness and size of Animat agent clusters. A visualisation of how the blocks can be arranged is seen in Figure 4.4, where each block is adjusted to maintain a small or part of an Animat agent cluster. An adjusted arrangement attempts to distribute large clusters over multiple blocks that are spatially decomposed with the scheduling of SDD blocks for update described in Chapter 4.3.

Figure 4.5 visualises the data structure composition of the SDD blocks. The environment and Animat storage components described in Chapter 3 are decomposed spatially. Each SDD block contains a multi-set of Animat agent pointers. Each SDD block also contains an array of grid cells and each grid cell contains a linked-list of pointers to agents that occupy the cell.

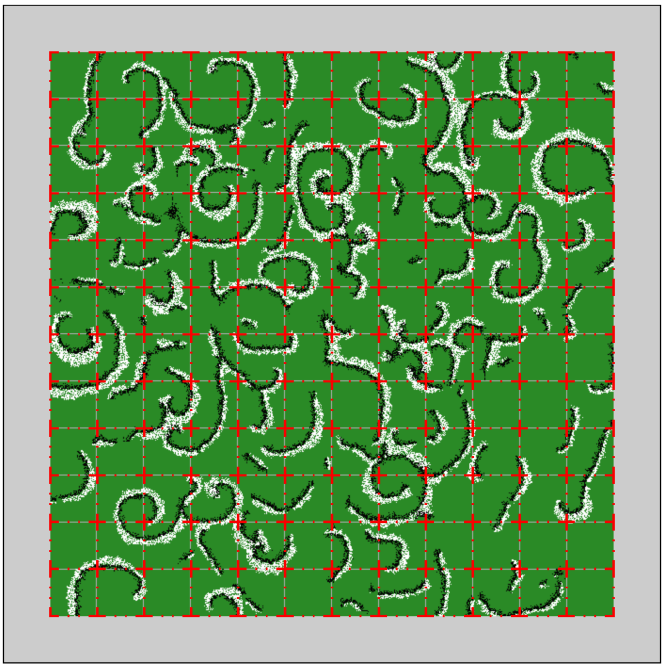

Figure 4.4: Spatially decomposed blocks that may contain portions of clustered Animat agents coloured white and black.

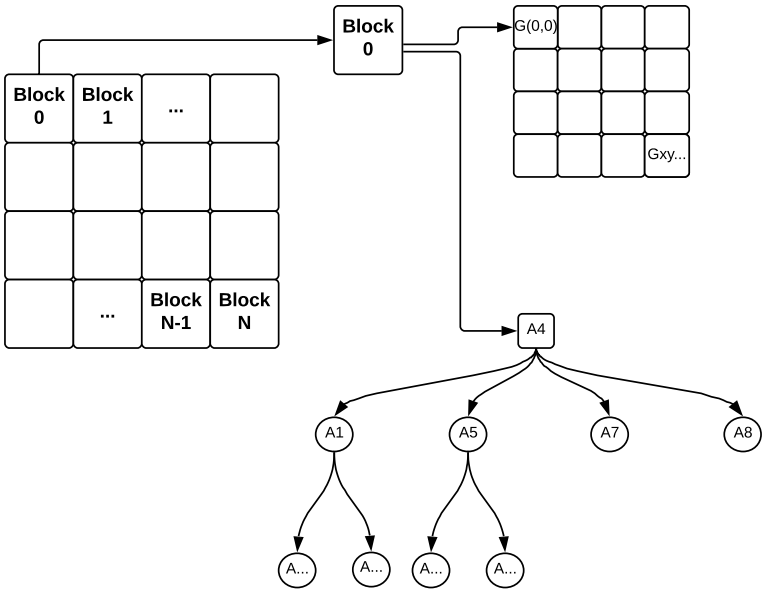

Figure 4.5: Spatially partitioned SDD block-0 that contains a multi-set of Animat agent pointers labelled *A*, and an associated array of linked-lists that contain the occupying Animat agents in each cell labelled *G*.

The implementation of the data-structure consists of C++ class objects that contains a N species array of STL multi-set of type Animat agent pointers $std :: multiset < Agent*, compare\_id > MSetAgents[N]$;. The $compare\_id$ can be replaced by any other comparison structure such as $compare\_xy$ which would insert Animat agent pointers in sorted coordinate order. The environment storage is also spatially decomposed and stored in the corresponding block of the SDD.

Listing 4.1 is an example block class with the Animat agent multi-set and an environment grid box array called search grid represented as private members. The integer dimension variables are for initialisation and offset calculations for access to each SDD block. The public members contain the constructors, overloaded operators, maintenance methods and access functions. Maintenance operations handle Animat agents pointers that have been created, marked for destruction or moved to another block. Access functions have pre-calculated offset values based on the dimensions of the block which return the address of the containers in the private section of the class. The update order of Animat agents in each block requires a random order; this can be achieved by creating a vector copy of the multi-set then using the same shuffle algorithms described in Chapter 3 to permute the order.

Listing 4.1: C++ Block Class

```
class SDDBlock{
private:

   int index, width, height; // Dimensions of SDD Block
   int min_x, min_y, max_x, max_y; // Bounds for block
   int search_grid_offset_x, search_grid_offset_y; // Grid Index Offset
   std::multiset<Agent*, compare_id> MSetAgents[2]; // Multiset Animat storage
   BlockGrid SearchGrid[block_width][block_height]; // Environment storage
   std::vector<Agent*> UpdateVector; //Optional to be in Block or Task Structure.
public:
   Block();
   Block(int index, int width, int height, int min_x, int min_y): index(index), width(width
       ), height(height), min_x(min_x), min_y(min_y){}
   Block(const Block& other);
   Block(Block&& other);

   Block& operator=(const Block &other);
   Block& operator=(Block &&other);

   ~Block(){};
   bool agent_out_of_bounds(Agent *agent);
   void block_add_agent(Agent *add_agent);
   void block_remove_agent(Agent *rem_agent);
   ... // Other member functions
};
```

4.3 Parallel Implementation

This section introduces the common parallel paradigms, API or framework and how they can be used to implement a parallel update strategy using the SDD approach. The SDD approach decreases the use of mutual exclusions and synchronisation by scheduling tasks in a non-neighbouring pattern.

The processing of tasks is also maintained in a balanced order where blocks that require the most processing time are scheduled first if possible. A summarised description and uses of the parallel paradigms, API or frameworks are as follows:

An execution model Pthreads can be utilised in the C/C++ language by including the **pthread.h** header file in the source code. The categories of low-level thread control available in pthreads are thread management, mutual exclusions (mutexes), conditional variables and synchronisation. One hundred or more procedures belong to these categories.

OpenMP is a high-level language that can be used in C/C++ and other languages such as Fortran by providing the **#pragma** directives to the compiler. A main feature of the OpenMP API is the loop construct **#pragma omp for [clause[[,] clause] ...] new-line for loops**. The construct specifies that the iterations of the loop or loops will be executed in parallel by the available threads. The iterations are distributed across the threads. This API, however, lacks thread control at a lower level.

MPI is targeted for distributed computing problems. The latest versions of MPI focuses on the use of shared memory and avoids some copying of memory between processes of a node. The library of routines can be used to parallelise the computationally intensive portions of a program. The primary communication functions are **MPI_Send** and **MPI_Recv** which are used to send and receive messages between nodes. MPI is often combined with other thread level interfaces such as OpenMP to achieve distributed, and thread managed parallelisable programming.

TBB is a C++ template library developed and maintained by Intel. TBB aims for the user to specify logical parallelism instead of threads, the library features automatically mapped tasks to threads at a high-level use. TBB relies on generic programming; thus libraries were developed for the C++ language to write the best possible algorithms with the fewest constraints. TBB coexists with other threading framework or packages as the library is not designed to address all threading or parallelisation problems.

CILK Plus or CILK++ originally developed by MIT and later acquired by Intel is an extension to the C and C++ languages to provide low and high-level parallelisation features. The main features are the efficient work-stealing scheduler for optimal use of threads, the vector parallelism support and powerful hyper-objects that enable lock-free design. The Animat multi-core implementation is developed with a combination of CILK Plus and C++ parallel features. CILK Plus is chosen over pthreads alone as it offers high-level and optimal scheduling features with low-level thread control. OpenMP is high-level only while MPI is targeted to distributed systems. The features of CILK Plus that simplify low and high-level data and task parallelism are as follows:

- `cilk_for` - Loop parallelisation.
- `cilk_spawn` - Expresses opportunities that a function can execute in parallel.
- `cilk_sync` - Specifies that all spawned calls or threads must complete before execution continues.

4.3.1 CILK Plus Worker Thread Mapping

CILK Plus is a suitable API to implement the multi-core Implementation of a typical Animat model in this thesis as CILK Plus provides low and high-level parallelisation features for development. Parallelism is achieved using an *N* number CILK Plus worker threads which are set by modifying the environment variable `CILK_NWORKERS`. A four threaded system for the Animat implementation on a Linux operating system requires a declaration `export CILK_NWORKERS=4`. The worker threads can be called upon using the `cilk_spawn` statement preceding a child function call. The `cilk_spawn` statement may execute the child function in parallel, it does not command parallelism but permits it.

The number of worker threads available is based on the hardware and configuration of the user. The user may also specify a lower number of threads to use than there are cores available. Using the SDD system, a thread is assigned to a block for processing. This system is not free of potential conflicts as simultaneous updates of neighbouring blocks still pose a risk for race conditions. A race condition can occur in the case of two agents being updated on the border of neighbouring blocks that are within one unit of each other. An example scenario is when a predator in one block attempts to eat prey that is in the neighbouring block. The issue may arise when another predator attempts to consume that same prey. There is no guarantee that the predator would have eaten the prey changing the status to deceased before another predator can eat that prey and result in the consumption of extra nutrients that do not exist.

It is possible to solve the race conditions by using mutual exclusions (mutex) locking mechanisms to protect shared data; however, it is well known that too many locks may negate the performance gains of a parallel system. In order to minimise mutex operations the blocks are updated such that no two neighbouring blocks are updated simultaneously. This ensures no conflicts occur such as two predators trying to eat the same prey simultaneously. If the size of the blocks are greater than agent vision range then only the nearest 8 blocks need to be considered as shown in Figure 4.6. Animat agents can see other agents within in their vision range. Ensuring that any neighbouring block cannot be updated means it won't simultaneously change. This allows any other blocks to be safely updated in parallel.

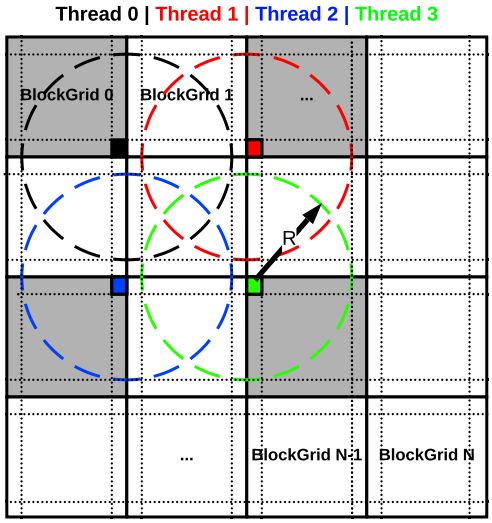

Figure 4.6: Non-neighbouring block update pattern by 4 threads with block size set to one unit more than the maximum vision radius of the Animat agent with the largest vision. The vision range set by radius R shows that each Animat agent cannot see any other agent currently being updated.

4.3.2 Task Scheduling

A task or SDD block scheduler is required to achieve the thread access patterns seen in Figure 4.6. The simple task scheduler developed in this thesis uses a combination of C++ conditional variables and mutexes to control memory access by CILK Plus threads. Tasks are initialised using the function in Listing 4.2 that clears the task queues each time-step then prepares the Animat agent update vector by inserting the pointers for shuffling.

After a randomised order of agent updates is provided, the rest of the task information can be set and written. Every task can be initialised simultaneously by using the `cilk_for` keyword to specify that the loop can be parallelised. Memory management is still required when inserting the tasks into the queue as conflicts will arise when two threads attempt to do this simultaneously. The function uses a mutex to lock the task queue exclusively for the thread that successfully attempts to access the queue memory.

Listing 4.2: Initialise Task

```
1  void init_sdd_tasks_CILK_Plus(){
2     TaskIQueueP1.clear();
3     TaskIQueueP2.clear();
4     tbb::mutex m;
5     cilk_for(int i = 0; i < n_blocks; ++i){ // Attempt to do in parallel
6        if(!UpdateVector[i].empty()){
7           UpdateVector[i].clear();
8        }
9        for(int s = 0; s < n_species; ++s){
10          AgentSet *agent_set = &Blocks[i]->get_agent_set()[s];
11          for(AgentSet::iterator it = agent_set->begin(); it != agent_set->end(); ++it){
12             UpdateVector[i].push_back(*it);
13          }
14       }
15       if(!UpdateVector[i].empty()){
16          randomise_update_vector(__cilkrts_get_worker_number(), i);
17       }
18       TaskSched *task = new TaskSched;
19       task->index = i;
20       ... // Set other task information
21       m.lock(); // Lock when adding task
22       TaskIQueueP1.insert(task);
23       m.unlock();
24    }
25    TaskIQueueP2 = TaskIQueueP1;
26 }
```

The task structure instances are stored in a multi-set using a custom compare operator to order the tasks based on the size or number of Animat agents in the corresponding SDD block. The task/block scheduler dynamically allocates tasks to each thread; however, the condition that the given task is not a neighbour of another task that is currently being updated must be satisfied before it is allocated for update. Figure 4.6 shows a non-neighbouring update pattern. To achieve this form of controlled thread execution, mutexes from the STL library can be used. The mutex is a primitive that is owned by a thread if it can successfully call the lock function, the ownership is released once the unlock method is executed. The CILK Plus implementation for the task/SDD block scheduler is seen in Listing 4.3. As this scheduler is executed using the `cilk_spawn` thread control feature, the work stealing scheduler that is automated in the `cilk_for` loop is not applied.

The following description of the SDD block scheduler refers to Listing 4.3. The mutexes: `mutex_TIQ`, `mutex_BIU` and `mutex_condition_var` seen in line 1 of Listing 4.3 are synchronisation primitives used to protect shared data from simultaneous access by multiple threads. The `mutex_TIQ` protects the task queue, `mutex_BIU` protects the data in a task and `mutex_condition_var` protects a conditional variable. A conditional variable is a synchronisation primitive that is used to block multiple threads at the same time until another thread modifies a shared variable such as `thread_wait_work` and notifies the conditional variable (`condition_var`). The declaration of the conditional and shared variable is shown in lines 2 and 3 of Listing 4.3.

The SDD block scheduler can update a preset number of agents for each block until all agents are updated in a simulation step. This is done by calling the SDD block scheduler multiple times until all agents are updated. The `int` parameters `partial_done` and `u_counter` are counters that are updated each time the SDD block schedulers is executed. When the `partial_done` counter equals the number of blocks then this means all agents have been updated. The `u_counter` variable is used

to ensure that each block is not updated more than once in the same iteration, this is seen in line 18 of Listing 4.3.

In the SDD block scheduler, the CILK Plus threads will in any order try to own the mutex to look for a task. During this task allocation stage each thread will iterate through the multi-set of tasks and check if the current task is safe to update. A task is safe to update if it is not a neighbour of another task that is currently being updated. Each thread will also check if the task has already been updated. If an updatable task is found, the thread will acquire the task by setting `curr_task = iter`. This task search stage is seen in lines 14 - 26 of Listing 4.3.

There is a possibility a thread will have failed to acquire a task to update, and in this case, the thread will enter the section of code that will cause it to wait by blocking as seen in lines 36 - 45 of Listing 4.3. In this critical stage the `condition_variable` blocks multiple threads at the same time until another thread modifies the shared variable `thread_wait_work` and notifies the `condition_variable`. The shared variable `thread_wait_work` signals if threads are waiting for work. The shared variable is initially set to false until a thread successfully locks the condition mutex and thereafter it is modified to true. The thread will now enter a waiting phase by calling the wait function on the locked mutex from the `condition_variable` in-turn atomically releasing the mutex. The `condition_variable` is seen in lines 31, 42, 67 and 74 of Listing 4.3.

This conditional variable can also execute a `notify_all()` function that wakes all threads that have been blocked. This is used to wake up any waiting threads after a task has been updated as seen in lines 64 - 68 of Listing 4.3. The update of a task occurs in lines 47 - 61 of Listing 4.3 where the thread that is allocated a task will update the task and safely deduct 1 from the count of total un-updated tasks.

There is a possibility a thread might have entered the wait phase just after another thread has notified all threads that it has completed the last task. To avoid this deadlock scenario, any threads that are waiting are woken to check again if there are tasks left. This safety mechanism occurs in lines 70 - 74 of Listing 4.3 in which the `condition_variable` calls a `notify_all()` function to wake waiting threads.

Listing 4.3: SDD Block Scheduler

```
mutex mutex_TIQ,  mutex_BIU, mutex_condition_var;
condition_variable condition_var;
bool thread_wait_work = false;
... // Other shared global variables
void sched_sdd_tasks_Phase1(int &partial_done, int u_counter, int &tasks_done, int
     rng_index, int thread_index, TaskSet *TaskIQueue,std::set<int> *NeighbourIndices ){
   bool tasks_left = true;

   while(tasks_left){
      int valid_task_index = -1;
      TaskSet::iterator curr_task = TaskIQueue->end();
      if(partial_done > 0){
         // Attempt to acquire non neighbouring task from multiset
         mutex_TIQ.lock();
         for(TaskSet::iterator iter = TaskIQueue->begin(); iter != TaskIQueue->end(); ++
              iter){
            int task_index = (*iter)->index;

            // If task is safe to update and there are remaining Animat agents to process
            if(!task_neighbour_conflict(&NeighbourIndices[task_index]) && ((*iter)->
                 all_agents_updated != true) && ((*iter)->update_counter == u_counter)){
               (*iter)->update_counter++;
               partial_done--;
               BlocksInUse[thread_index] = task_index;
               valid_task_index = task_index;
               curr_task = iter;
               break;
            }
         }
         if(partial_done == 0){
            std::unique_lock<std::mutex> unique_lock(mutex_condition_var);
            thread_wait_work = false;
            tasks_left = false;
            condition_var.notify_all(); // Signal to other threads if they are waiting
         }
         mutex_TIQ.unlock(); // Allow other threads to find work

         // If thread couldn't find valid work then wait for signal to try again
         if(curr_task == TaskIQueue->end()){
            std::unique_lock<std::mutex> unique_lock(mutex_condition_var);
            if(partial_done != 0){
               thread_wait_work = true;
            }
            while(thread_wait_work){ //keep thread waiting
               condition_var.wait(unique_lock);
            }
            continue;
         }
         // Update Task
         else if(curr_task != TaskIQueue->end()){

            int current_update_index = (*curr_task)->uv_update_index;
            int n_agents = (*curr_task)->agents_per_block;
            bool completed = update_n_agents_phase_1(valid_task_index, rng_index,
                 current_update_index, current_update_index + n_agents);
            (*curr_task)->uv_update_index = current_update_index + n_agents;

            mutex_BIU.lock();
            BlocksInUse[thread_index] = -1;
            if(completed){
               (*curr_task)->all_agents_updated = true;
               tasks_done--;
            }
```

```
60              mutex_BIU.unlock();
61          }
62
63          // Signal threads that are waiting to re-attempt to search for tasks
64          if(thread_wait_work){
65              std::unique_lock<std::mutex> unique_lock(mutex_condition_var);
66              thread_wait_work = false;
67              condition_var.notify_all();
68          }
69      }
70      else{ // Notify all threads that no work is left and to return
71          std::unique_lock<std::mutex> unique_lock(mutex_condition_var);
72          thread_wait_work = false;
73          tasks_left = false;
74          condition_var.notify_all();
75      }
76  }
77 }
```

Scheduling tasks for phase two differs slightly from phase one as memory conflicts could occur during the update of the multi-set Animat storage and environment search grid. The scenario where two Animat agents moving from two different neighbouring blocks to the same new block could cause issues if the two Animat agents are inserted at the same time. This, of course, can be solved using mutex locks; however, another solution is to schedule tasks that are two blocks apart enabling a lock free approach.

The scheduling of tasks is similar to the phase one seen in lines 14 to 26 of Listing 4.3. The difference is that the block of a task can only be updated if they are two blocks away from another block that is currently being updated. This avoids race conditions as any agents that are on the borders of any block that is updated can only move to any neighbouring block, so updating blocks that are two apart ensures that no other blocks that are simultaneously updated share the same neighbours. This means the storage of blocks that are neighbours to a block that is currently being updated is not required to be locked for insert or remove operations.

The same sequence of mutex locking is used to safely access the multi-set of tasks to search for a task to update. This approach effectively allows updatable blocks to access neighbouring blocks to process movement of agents without having to lock associated multi-sets of Animat agents and the array of linked-lists described in Figure 4.5 The pattern of two neighbouring blocks apart is seen in Figure 4.7

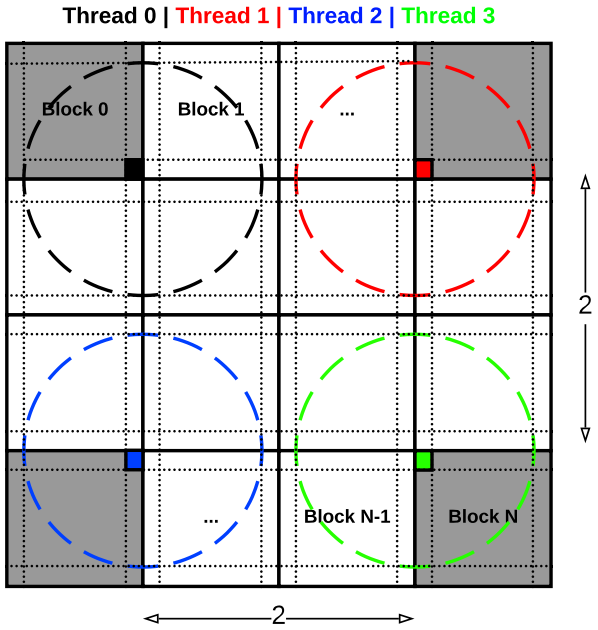

Figure 4.7: Phase two is updated by scheduling blocks that are a minimum of two blocks apart so that no blocks that are simultaenously updated share any neighbours. This allows blocks that are currently being updated to move agents on the borders to neighbouring blocks without having to lock the storage in the neighbouring blocks.

4.3.3 Load Balancing

Scheduling tasks with no form of distribution management could add unnecessary overheads such as the increase of total execution time due to the order of tasks processed. In this Animat multi-core implementation, simple load balancing is used in which the blocks with the most agents are updated first. If the tasks are updated simultaneously and the largest task is scheduled to be updated last then all other threads that have completed their smaller tasks may have to wait while only 1 thread updates that largest task. This situation typically results in a longer processing time of all tasks when compared to load balancing by distributing larger tasks first.

Distributing largest work loads first is a simple solution to maximise efficiency for the scheduling of tasks to update SDD blocks. There is a possibility that biased update orders are introduced if all Animat agents in the largest tasks are always updated first. To solve this problem an iterative process is implemented where an N number of Animat agents each block is updated per block for all blocks in each iteration. The iterative process ends when all Animat agents have been updated. Figure 4.8 is an example of the iterative update of a 4x4 block system where 10 Animat agents are updated in each block per iteration.

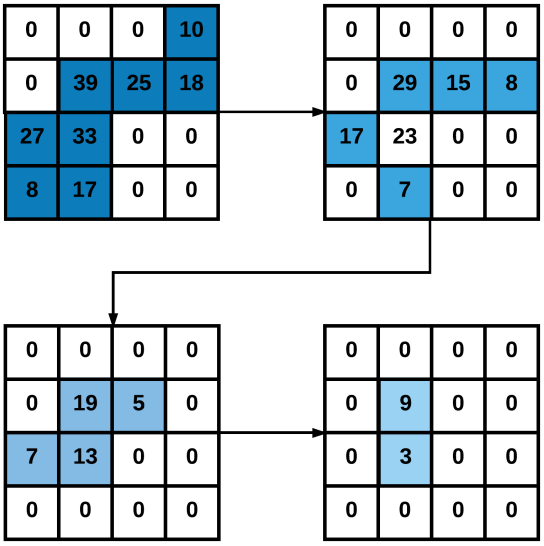

Figure 4.8: Ten Animat agents from each block coloured blue are updated in each iteration with decreasing counts of remaining agents for processing.

4.4 Summary

Experiments with larger model sizes of a typical Animat model may require more computing power to simulate or trial numerous parameters in a practical time frame. This Chapter identifies the dependencies of the Animat model and offers an approach to resolve parallel issues to develop an implementation strategy. The SDD storage combined with a managed update strategy utilises the CILK Plus constructs combined with C++ synchronisation and thread control to implement the Animat model on multi-core architecture. The `cilk_for` construct is used for the initialisation of the SDD tasks while the `cilk_spawn` and `cilk_sync` are used to schedule these tasks for update.

This implementation provides sufficient speed-up of simulation updates for real-time visualisation and analysis. Other update strategies and data structures different from the proposed structures and methods in this chapter can be used to implement a multi-core Animat model. However, the focus of this thesis is on the GPU architecture and the analysis such as model dependencies can be considered when developing a solution to simulate the model on GPU architectures. Model sizes that can support millions of Animat agents and complicated modification or extensions to the Animat model may require greater computation power. Chapter 5 offers some strategies and an implementation of the Animat model on the GPU architecture.

5

GPU CUDA Implementation of the Animat Model

Implementing the typical Animat model on a many-core GPU architecture may help to improve performance to explore various arrangements of the model parameters on larger scale model sizes that may not be time-wise, feasible, on single and multi-core implementations discussed in Chapter 3 and 4. The performance increase of an efficient multi-core implementation may only improve a speed up of close to N number of cores available which can be easily outperformed by a GPU released within a relative time-frame.

The GPU hardware combined with the CUDA platform can provide greater computation power required to explore larger model sizes compared to single and multi-core implementations. GPU's can effectively support real-time analysis of experiments that result in agent populations growing to millions.

Dependencies across parallel implementations are similar; however, the computational method to update the simulation differs due to the execution model of each architecture. To implement a typical Animat simulation using CUDA on GPUs, a parallel strategy must be developed to support the requirements of the model on the Single-Instruction Multiple-Thread (SIMT) execution model whilst carefully considering the CUDA programming model to achieve high performance. A suitable data structure must be developed to support the spatial aspect of the model that can be arranged to suit the requirements of optimal access patterns on various types of GPU memory.

A key requirement of parallelising the Animat model is to retain the original model behaviour. This can be confirmed by matching agent states on the GPU to the serial CPU implementation over an adequate number of steps. Matching of the states confirm that using the same random number generator and seed on both implementations will result in the same simulation sequence. This replication process is discussed in Section 6.4. To replicate original model behaviour, a concurrent update of Animat agents are required to follow a strict order and conflicts caused by agent interactions need to be resolved to allocate limited resources to prioritised individuals correctly.

5.1 CUDA Platform

CUDA is a parallel programming platform and model developed by NVIDIA for general purpose computing using GPUs. CUDA programs are written in C/C++, Fortran or other supported programming languages. CUDA is used to decompose parallelisable portions of a program into fine grain tasks that can be assigned to threads to instruct simultaneous computation by many streaming multiprocessors of the GPU.

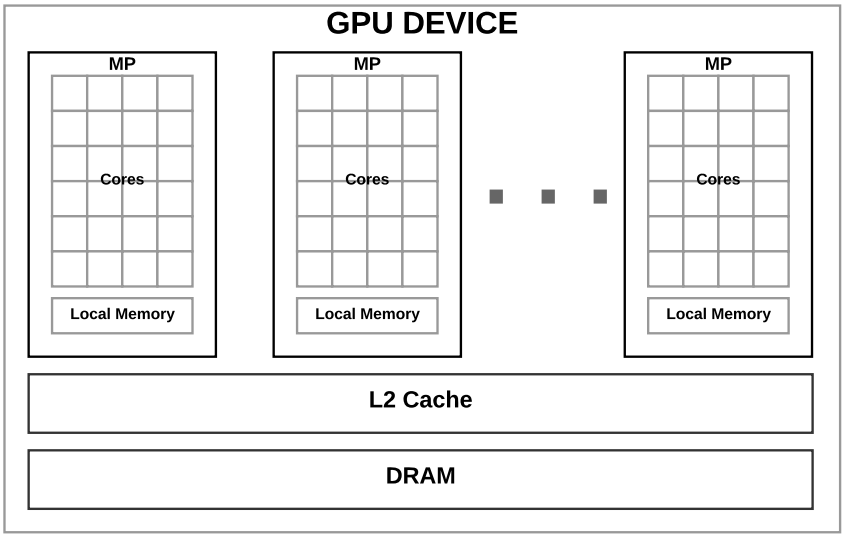

Figure 5.1: GPU device with many Streaming Multiprocessors that each can access the device memory (DRAM) through the L2 cache

Figure 5.2 Shows the NVIDIA Kepler streaming multiprocessor architecture with 192 cores. Each streaming multiprocessor has quick on-chip memory such as the registers, L1 data cache/shared. The device has many streaming multiprocessors which can access the slower but larger DRAM through the L2 cache as shown in Figure 5.1.

The term thread warp in CUDA is a basic unit of execution which consists of 32 threads that execute the same instruction. Thread blocks in CUDA refers to a group of threads consisting of many warps that can execute in parallel. Each block is executed on one multiprocessor which allows the threads of a block to access the shared memory of the multiprocessor. For the Kepler architecture, each multiprocessor has a maximum of 2048 threads and can concurrently process 64 warps or 16 blocks. A kernel grid in CUDA consists of a set of thread blocks. The kernel grid is executed by the multiprocessors of a GPU device. Figure 5.3 illustrates the mapping of a 2-dimensional grid that consists of many 3-dimensional blocks and each block consists of many warps. This example can be convenient to map the threads to a 3-dimensional data-set.

Threads are scheduled by the platform, and the coordination of threads are managed by the programmer. The programmer specifies how threads are grouped into a block by specifying the size of each of the three dimensions. A grid can have one to three dimensions consisting of many

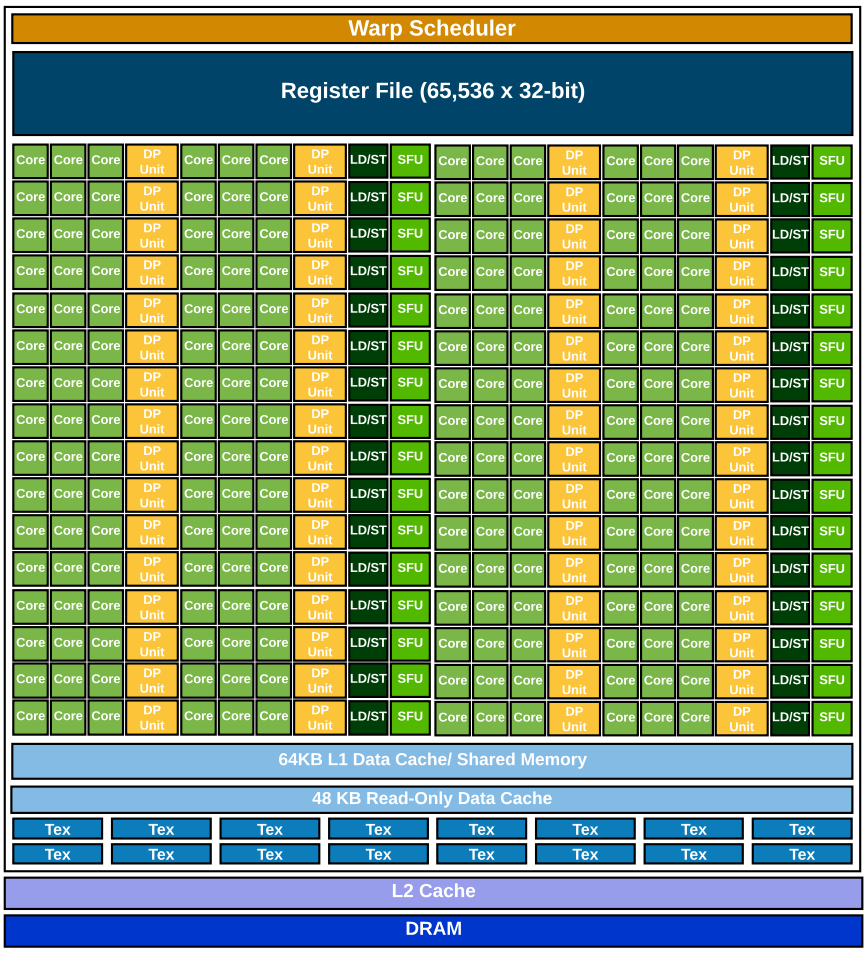

Figure 5.2: Streaming Multiprocessor of the NVIDIA Kepler Microarchitecture with the L2 cache and device memory (DRAM).

blocks that have unique identification numbers which can be used to calculate the identification of a single thread relative to the grid. Typically the grid and block dimensions are specified to allow enough threads to cover the data-set however it is possible to set a lower number of blocks and use a grid-striding loop to iteratively compute unprocessed data.

Threads can cooperate and share data using shared memory, and this memory is allocated per block which means only threads of the block can access the allocated shared memory. Shared memory is typically protected with thread barriers to ensure the synchronisation of threads to wait at a certain point until all threads have performed their instructions to reach the barrier and thus preventing reads of shared memory that may have been modified.

The launch of a global CUDA kernel is managed by the thread block scheduler to dispatch threads blocks to multiprocessors. The warps of a thread block becomes active when allocated and are placed in a pool for a single or dual-issue warp scheduler to select and issue instructions. This execution model is known as SIMT where multiple independent threads execute a single-instruction

concurrently.

The control flow of thread execution paths are coordinated by the programmer implemented as CUDA kernels. Ideally, all threads in a warp have the same execution path when no conditional branching is required, resulting in optimal utilisation of execution units. Different execution paths occur when threads diverge as a result of branch conditions; this is referred to as divergent branching. Divergent branching can be an issue as threads diverged to other paths that must be executed serially until the paths re-converge. The modern Volta architecture alleviates some performance issues of thread divergence by allowing forward progression of diverged threads.

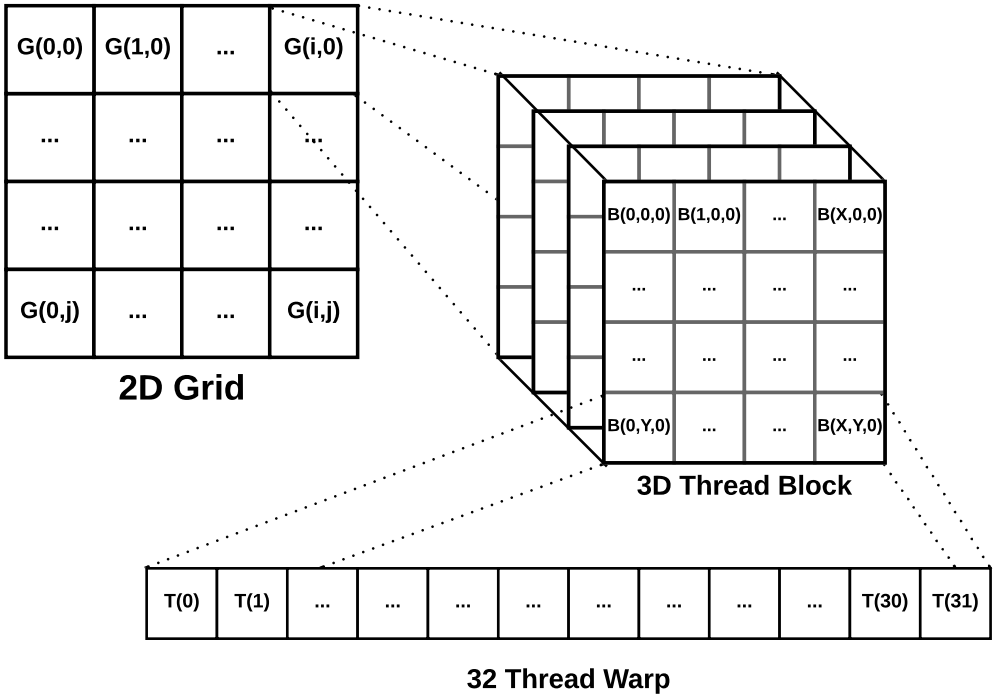

Figure 5.3: Thread hierarchy in CUDA programming: warps of a 3D block and blocks of a 2D grid where T(x) is the thread index of a size 32 warp and B(x, y, z) is the thread index for each of the 3 dimensions of the block. G(i, j) Is the block index for each of the 2 dimensions of a grid.

5.2 CUDA Template Libraries

Parallel primitives such as sort, scan and reduce can be written to support application requirements. The generic nature of these algorithms has lead researchers to develop highly optimised and reusable libraries to support the development of sophisticated algorithms or applications.

CUDA template libraries provide common parallel algorithms and data structures that can be used through high-level interfaces for rapid prototyping and high-performance software components for production. The two libraries used in this thesis are Thrust and CUDA UnBound (CUB).

Both these libraries are similar as they provide device-wide primitives for the CUDA application. They target different abstraction layers with CUB providing an interface that is lower-level than thrust and includes a library of SIMT collective primitives for block and warp-wide programming. Thrust is typically used for rapid prototyping of CUDA applications to test implementation details and CUB for production release. Thrust's backend is built on top of CUB which is a project of the NVIDIAs research team.

5.2.1 CUDA UnBound

The CUB library is developed by Duane Merrill of NVIDIA research. This library provides highly optimised generic primitives that target various levels of development such as parallel sort, scan and reduce. These parallel primitives are available for all the levels of the CUDA programming model which is warp-wide, block-wide and device-wide. The warp-wide level means that these functions are computed on items partitioned across a CUDA thread warp. The functions of a block-wide level are computed on items partitioned across a CUDA thread block. Device-wide functions are computed on data items that are stored in device memory.

The warp-wide collection includes warp-scan, warp-reduce and warp-lane shuffle functions. The block-wide collection provides more primitive classes and the common methods include block-reduce, block-scan and block-radix-sort. Device-wide primitives target a single or segmented batch of problems, and they operate on data that resides within device-accessible memory and include device-radix-sort, device-reduce and device-scan. These are CUDA implementations of parallel radix-sort, reduce and scan.

5.2.2 Thrust

Thrust is a high-level template library based on the C++ standard template library that can help to implement parallel applications and can be fully interoperable with the CUDA parallel computing platform.

The thrust template library works across different architectures providing high-level algorithms for developers. The fundamental parallel algorithms such as scan, sort, and reduction are managed by the library. Key features of the thrust template library include the STL-like data structures and parallel primitive algorithms. Key thrust data structures include host vectors, device vectors, device pointers and key algorithms include sort, reduce and exclusive scan. Thrusts backend uses CUB methods and manages the memory requirements of the primitives which may induce extra overhead of repeated memory allocations if used iteratively.

5.3 Parallelisation Strategies

The parallelisation strategies are designed to develop the Animat model using the CUDA platform to harness the computational power of the GPU hardware while maintaining the original Animat model behaviour. The strategy is planned to execute on the GPU architecture which requires a strict ordering system to correctly update and allocate resources to reproduce original model behaviour.

This section discusses the strategies to satisfy the requirements of parallelising parts of an Animat update cycle.

Updating Animat agents one at a time in a specific order is not an ideal solution for a GPU implementation. The preferred approach is to update all Animat agents simultaneously whilst retaining the original model behaviour. This can be achieved by assigning each Animat agent a priority number which represents their position in the relative sequential update cycle. This means that agents with a lower priority number acts before agents with higher numbers. Each Animat agent optimistically makes a decision about the actions they are going to take, which may be invalid if the action conflicts with another agent with a lower priority number. These conflicts must be resolved in additional stages. Once the Animat agent decisions are made, the actions are performed with consideration of other agent priority numbers. This is to ensure that the outcome of the actions are based on a particular order.

Figure 5.4 shows the update cycle of the typical Animat model on a GPU architecture with a generation of priority numbers performed to support the decide and perform phase updates of the model. A description and use of the priority number system is discussed in the subsection that follows. Post-processing of the data is performed last to correctly rearrange data as a result of Animat agent updates. Post-processing a requirement of the support data structures. Optional processes include pre-processing which functions as optimisations to minimise re-calculations of required information and visualisation to analyse data in real time.

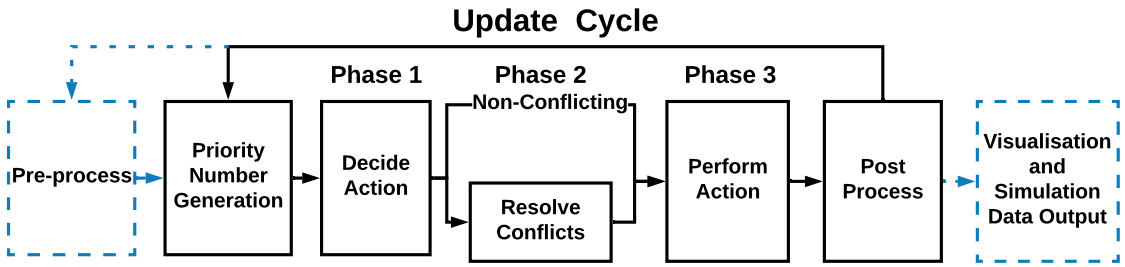

Figure 5.4: The processes of an update cycle where priority number generation is performed to support the phase updates. After the phase updates the data structures are post-processed to correctly rearrange data for the next cycle. Pre-processing and Visualisation marked with blue are optional.

5.3.1 Priority Number

The Priority number (PN) system can be used to replicate the update order on a sequential implementation of the Animat model. A serial implementation of the Animat model updates all agents each cycle in a random sequence and each at a time. A priority number is a unique value that is assigned to Animat agents to provide that sequence during a concurrent update of the agents. This allows the agents to be correctly updated and resources to be correctly allocated to match the original model behaviour.

Priority numbers support the decision process by allocating some unique value to each indi-

vidual which signifies their decision priority. For example, if there are two Animat agents, (A) and (B), If (A) is to complete a phase of their update before (B), then an integer value (1) can be assigned to (A) and (2) to (B) etc. The value can be of any sensible type provided they are unique and conform to some order. Animat agents therefore can base their decisions on other agents PN and decision. When Animat agents with lower PN make their decision, they are not influenced by other agents with higher priority numbers. The implementation of PN and allocation process is described in Section 5.5.

5.3.2 Decision Stage

The decisions of Animat agents may depend on the actions of other agents that act before them. This is simple to implement in serial as updating agents in order of priority will provide all necessary information to agents that are updated afterwards. When all Animat agents are updated in parallel the actions of agents with higher priority order may not yet have been performed. On a GPU architecture, thousands of cores and threads are available to perform these tasks simultaneously, but a method must be implemented to give the same behaviour.

The decision stage involves many agents deciding on a behaviour that is based on their internal state and the situation of their individual environment. These factors influence their decision to choose a rule from their strictly ordered rule set which is an intent that can be acknowledged by other agents. The decision stage can be asynchronous to a certain extent with all individuals deciding on what they want to do based on their rule set. Figure 5.5 represents the flow of an Animat agent's decision process where they can decide on a rule in priority order if the health toll and age update for the step does not cause their death first.

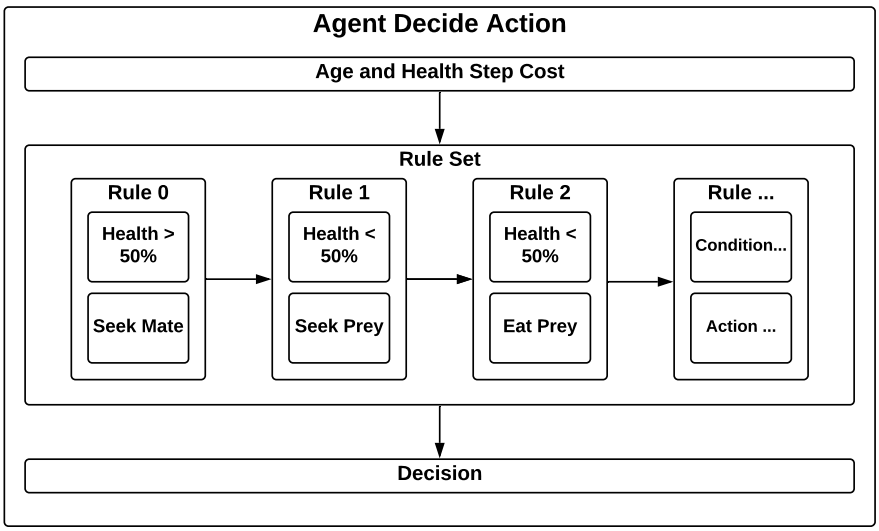

Figure 5.5: Animat agent decision process where an agent updates the health and age toll for a step at which only live agents proceed to decide on a prioritised rule if the relative condition is satisfied.

Decisions can be made independently or may depend on other Animat agents. Independent decisions include breeding, grazing and any movement rules, as the condition for these actions in a typical Animat model does not have dependencies. Dependencies for these rules do not exist as the information required to satisfy the conditions do not change until the end of the update cycle. Examples include the number of neighbouring Animat agents or the location of the closest threat. These decisions are also referred to as non-conflicting.

5.3.3 Conflicting Decision

Conflicting decisions refers to many Animat agents enquiring over limited resources. An example of conflicting decisions is when many predators express their desire to consume an individual neighbouring prey. This wouldn't be an issue in a sequential update on a CPU as the predator updated first would have rendered the prey deceased due to consumption. The next predator due for an update that is within interaction range of the consumed prey acknowledges that prey as being consumed and may choose another prey within range or move on to the next rule of the ordered rule set.

This model behaviour is potentially conflicting if all Animat agents are to be processed simultaneously. The PN system can support a process to resolve these conflicts by providing the sequence from the original model which determines the priority of Animat agents and the resources they may acquire. A simple example using the PN system to resolve a conflict is as follows, consider two neighbouring Animat predators (A) with PN (5) and (B) with PN (3) and their desire to consume prey (C) with PN (7). Predators (A) and (B) checks if prey (C) is alive and when deciding to consume prey (C) their priority numbers indicate predator (B) has priority, thus (A) may have to decide to search for another prey or attempt another rule. The previous example is a simple case, a general case where the model has reached standard density may include hundreds or more predators enquiring over the limited number of prey around them.

5.3.4 Performing Actions

Once all decisions are finalised, threads are scheduled to execute these decisions. All actions of the predator agents can be performed and updated before the prey. Although this sequence is not necessary it may help to avoid updating agents that die of circumstances such as predation. The PN system allows individuals of each species to determine the outcome of actions that may involve other agents by considering the order of actions defined by the PN.

An example is the interaction of predator and prey, on a sequential system if the prey is updated first and decides to graze then their health state has already changed which means the predator that consumes it afterwards will get more health. The PN system replicates this by pre-allocating a PN that has a lower priority for the prey thus when they are updated simultaneously the predator will consider the preys decision and obtain the health inclusive of grazing when performing the eat action.

In a typical Animat model, the health of newly bred agents is based on an average of their parents. This model configuration may cause the breeding action to have a dependence on the grazing or eating action of the partnering individual. In a case where an Animat agent has found a partner

to breed, but the partner decides to eat, and has a higher priority PN, then the health value of the partner after it has eaten must be used to calculate the health for the newly bred agent.

Movement decisions such as the move-towards, away and in a random direction do not cause conflicts as the movement of the Animat agents do not occur during the decision phase. Animat agents performing actions require no synchronisation, but they need to know the decisions and priority numbers of other agents to compute the out come of actions in an order that reproduces the original model behaviour.

5.3.5 Post-processing

Post-processing involves rearranging elements within the data structures developed to support the parallel update strategies discussed in this chapter. The data structure is discussed in section 5.4 and provides implementation details such as the ordering of elements in which post-processing is executed to prepare the data structure for the next update cycle.

5.3.6 Pre-processing

The Pre-processing stage is an optional optimisation for Animat on GPU. Many Animat agents may perform and repeat the same computation as other agents during the update cycle. For example, when two or more predators in the same cell search for the closest prey. This can be expensive to re-compute on the GPU as this may require extra memory operations to reload the required data. As this data does not change during update it can be computed once and re-used.

5.4 Data Structures

The decide and perform phase update strategy, requires a data structure designed for performance on GPU architectures. An important aspect of high performing CUDA applications is the design of data structures to support algorithms to achieve desirable memory transactions and utilisation. This section focuses on the storage requirements for a typical Animat model using the compressed sparse data structure, the helper structure and the lookup device memory arrays. A diagram of the storage requirements for the implementation of the Animat model can be seen in Figure 5.6.

The compressed sparse structure borrows the idea of the compressed sparse rows (CSR) data-structure in which only nonzero elements are stored with index arrays to lookup corresponding matrix positions. The Animat model can have more than one agent in each cell and may not be as sparse as some sparse matrices that is suitable for storage in CSR. There could be hundreds of Animat agents in a cell and using CSR would require duplicate indexing entries for each agent in the same cell. The compressed sparse structure stores the agents sorted by position with indexes for each grid cell. The indices provide the position of the first Animat agent in the storage and can be used to calculate the number of occupying agents by examining the next cell with an agent.

The compressed sparse structure consists of primitive type arrays to support coalesced memory access patterns and can be used as a single instance per species. The Animat agent and environment

storage can be combined into the compressed sparse structure to utilise an indexing system to provide efficient coalesced memory transaction for searches and direct indexing to Animat agents at each cell location. The helper structure is a single instance that supports both species of the compressed structure and includes data such as device arrays of random number generators, and the primitive type shared variables for assignment of unique identification and conflict resolution flags. Search lookup offsets are stored in a constant memory array with one per environment dimension and the species data is stored as an array of instances for each species. The species data can also be stored in the compressed sparse structure or kept separated if passed to device constant memory.

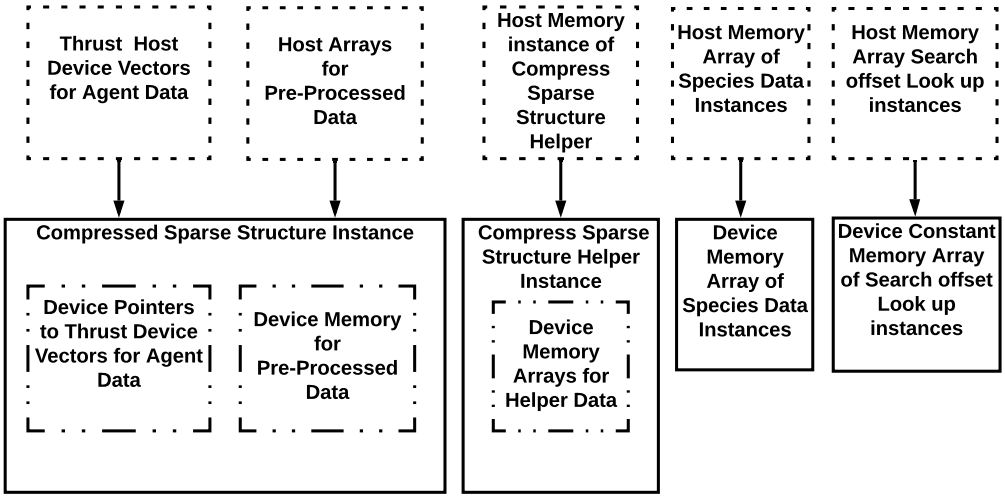

Figure 5.6: Illustration of the data structures with data for host memory in dashed squares and data for device memory in solid squares which are passed to CUDA kernels for access.

5.4.1 Compressed Sparse Structure

The compressed sparse structure consists of Thrust vectors which are based on the C++ Standard Template Library and can be interoperable with the CUDA parallel computing platform by allowing for raw pointers to be extracted for access on device kernels. Thrust host vectors store data on CPU memory and device vectors on global GPU memory. The generic Thrust host and device vectors work like C++ STL vectors and offer functionalities such as resizing and overloaded assignment operators to simplify the process of managing and exchanging data between the host and device vectors.

Thrust vectors provide additional iterators commonly referred to as integer pointers to the beginning and end of a specific range. The range iterators are used in the Thrust generic algorithms such as sort, generate and copy. These iterators can be zipped for the multiple agent attribute device vectors to create tuples in which all associated containers are sorted by the same key.

The high-level abstraction of Thrust primitives means the library manages the memory. Thrust functions automate the re-allocation of memory rather than re-use which may incur extra overhead.

The CUB library can be used in such cases provided a raw pointer is extracted from the device vectors for CUB methods to access. Although CUB does not offer high-level features such as zip iterators from Thrust, CUB provides better performance in parallel primitives such as sort, prefix scan, reduction, histogram [193].

The use of C++ classes in the CPU implementation offered encapsulation properties for data members of Animat agents. This is commonly referred to as an array of structures (AOS) and can be implemented on the GPU platform. Another common data structure is referred to as a structure of arrays (SOA). Both data structures serve the same purpose; however, they are implemented and accessed differently. Listings 5.1 and 5.2 shows how the AOS can be implemented compared to the SOA. The AOS container is an array of device Animat agent pointers or objects instances whereas the SOA is a structure of all device agent data containers in which each index belongs to an Animat agent.

Listing 5.1: Array of Structures

```
struct DeviceAgents{
   int location;
   int data_A;
   int id;
   int move_number[2];
   int age;
   int health;
   int eaten_by;
   char action;
   bool alive;
};
```

Listing 5.2: Structure of Arrays

```
struct DeviceAgents{
   thrust::device_vector<int> location;
   thrust::device_vector<int> data_A;
   thrust::device_vector<int> id;
   thrust::device_vector<int> move_number
       [2];
   thrust::device_vector<int> age;
   thrust::device_vector<int> health;
   thrust::device_vector<int> eaten_by;
   thrust::device_vector<char> action;
   thrust::device_vector<bool> alive;
};
```

Implementing AOS on CUDA for Animat agents can be achieved using the forced alignment of structures or explicitly aligned structures such as float4 or uint2 types that satisfy the memory guidelines. The AOS is useful when the data member is accessed simultaneously; thus the memory locations are contiguous allowing coalesced loads from global memory. The SOA is useful when the containers are accessed separately; for example, all of the device vector $data\_A$ is accessed before device vector id resulting in coalesced read or write transactions.

An illustration of threads accessing Animat agent data is seen in Figure 5.7 where threads of a warp require transactions to the K-index stored in global memory. The top diagram shows when using an AOS storage the thread access of the K-index results in unaligned memory transactions. When threads access a field of their agents, SOA stores the fields in sequential addresses. This means sequential threads access sequential addresses in memory allowing the transactions to be combined into a single coalesced transaction.

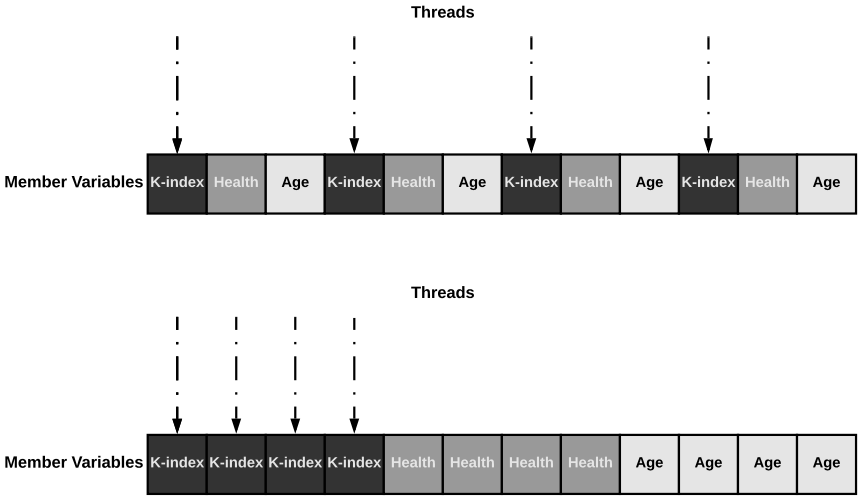

Figure 5.7: Access pattern of threads using AOS at the top and SOA at the bottom.

The compressed sparse structure is implemented as a C struct extending the SOA approach and can be referred to as the device state as seen in Listing 5.3. The device state consists of raw pointers, counters and post-processed data pointers. The raw pointers contain the address of the casted device vector iterator for use in CUDA kernels or CUB methods. The post-processed data pointers address allocated device memory that contains post-processed data for the next update cycle. The counters record information such as the current number of live agents or the current index to unallocated memory for new agents. Combining data members that are required throughout the update cycle simplifies development of CUDA kernels. Only a pointer to the device state is required to access any of the data members.

Listing 5.3: Compressed Sparse Data Structure

```
struct DeviceState{
   int live_agents;  // Number of live agents per update cycle
   int free_indices; // First index of contigous memory to add new agents
   // [Raw pointers casted from device vector iterators]
   int *d_location_ptr;       // K-index of 2-D position
   int *d_data_A_ptr;         // Temporary data  storage for each update cycle
   int *d_id_ptr;             // Identification number
   int *d_move_number_ptr[2]; // priority number
   int *d_age_ptr;            // Age
   int *d_health_ptr;         // Health
   int *d_eaten_by_ptr;       // Index to other predator consuming this agent
   char *d_action_ptr;        // Decision
   bool *d_alive_ptr;         // Status
   // Pre and post processed data
   int *d_start;           // Start index array
   int *d_cell_count;         // Count of occupying agents in cell locations
   int *d_neighbour_count;    // Count of neighbouring agents in cell locations
   int *d_first;           // Index to first agent in neighbourhood
   int *d_last;            // Index to last agent in neighbourhood
};
```

5.4.2 Animat Start Index

Animat agents may need to search their neighbourhood for information about other agents to make a decision during the decision stage. The start index array (SIA) is a part of the compressed sparse structure and can assist the search for other Animat agents. The SIA helps by enabling efficient look up of Animat agents at specific cell locations. This SIA works by providing the index of the first Animat agent in the Animat storage at each cell location. The SIA data can be easily computed as the Animat agents are stored in the compressed sparse structure using the SOA approach in a sorted order.

The SIA is a one-dimensional device pointer container that stores indices as integers that are reset to the value -1 at the start of each update cycle. To populate the SIA, a CUDA kernel is used to assign each thread in a block to each Animat agent. The position of the current and preceding agent is compared to check if they are not equal. When the positions are not equal the preceding agent is actually in another location thus the current agent is the first in its cell location, so the thread index that is mapped to this agent can be written to the SIA.

Figure 5.8 is an example outcome of using the CUDA kernel from Listing 5.4 to populate the SIA with indices to each first agent at each cell location. A value of -1 remain for any cell location that is not occupied. The number of Animat agents occupying each cell can also be calculated in this CUDA kernel to re-use the data loaded from global memory. This is stored in an agent cell count array and avoids the need for another kernel launch and reloads of global memory. It is possible to find the count of agents in a cell by comparing the value in the next cell. If the value in the next cell is different then the difference is the number of agents and this is seen in the CSIA array in Figure 5.8. The SIA or CSIA effectively provides direct indexing from the environment to the Animat storage enabling methodical indexing to various searching patterns.

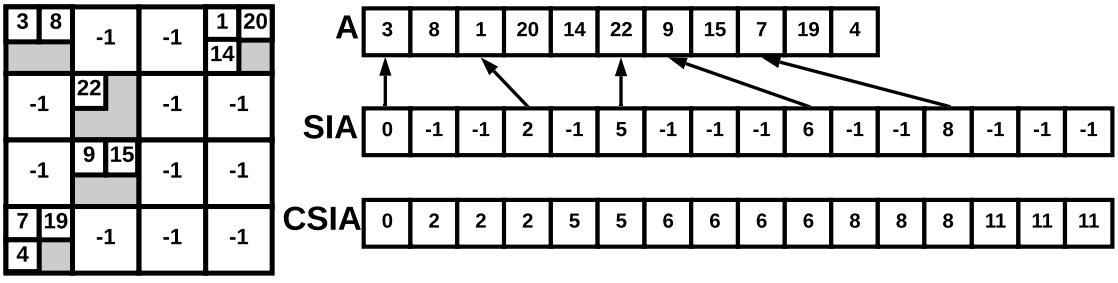

Figure 5.8: Example 4 by 4 Animat environment with agents and their unique identification number located in grey cells, The (SIA) contains the index to the first agent at each cell location in the Animat agent storage (A). The (A) stores Animat agents in a sorted order. The (CSIA) can be used in place of the (SIA) to also provide the number of agents in a cell by comparing the value in the next cell. If there is a difference then the difference in the values is the number of agents in that cell.

Listing 5.4: Set Start Index Algorithm

```
__global__ void  set_sia_cell_count(DeviceVecPtrSize AgentList, int *agent_start, int *
    agent_cell_count) {

   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   int vec_size = AgentList.live_agents;
   if(tid >= vec_size){
      return;
   }
   int agent_pos = AgentList.location_ptr[tid];
   int agent_pos_m1 = AgentList.location_ptr[(tid == 0) ? 0 : tid-1];
   // Compute index to first agent at cell location
   if(agent_pos != agent_pos_m1 || tid == 0) {
      agent_start[agent_pos] = tid;
      //check for last index
      if(tid == vec_size - 1){
         agent_cell_count[agent_pos] = 1;
         return;
      }
      int iter = tid;
      while(true){ // Tally occupying agents
         if(agent_pos != AgentList.location_ptr[iter]){
            break;
         }else{
            iter++;
         }
      }
      // Write number of occupied agents if any
      if( (iter - tid) > 0){
         agent_cell_count[agent_pos] = count;
      }
   }
}
```

5.4.3 Compressed Sparse Structure Helper Data

The helper data structure contains additional information that can be accessed by both the predator and prey species of a typical Animat model and can be stored in a separate structure as only a single instance is required. Helper data includes the following:

- Device array of random number generator states used for the simulation.
- Variable to allocate unique identification for each Animat agent that has existed in a simulation run.
- Variable to index the priority number array that is not being permuted if CUDA streams are used to shuffle priority numbers simultaneously.
- Device arrays of post-processed data such as indices to the first prey or predator in vision within the main Animat storage in the compressed sparse structure.

5.4.4 Search Offsets

The search offsets discussed in Chapter 3 can be used in the GPU implementation of the Animat model to directly index cells in the environment in a specific order. The offsets are stored in device constant memory as they do not change and are repeatedly access by threads that are assigned to Animat agents. The constant memory size is limited but cached meaning only cache misses cost a read from device memory. The offsets can also be stored in global memory; however, the cost of memory transactions may be expensive if the access pattern is undesirable.

The pre-calculated search offsets for the typical Animat configurations with predators having a search radius of 60 units will require 11,630 pairs of coordinates to be stored. Storing these as integers in constant memory may not fit on older device architectures with less capacity. These can be stored as `char` types on the device; however, configurations with a large vision radius may require too much device constant memory and must be stored in other memory. The device constant search offset arrays can be directly indexed and simply cast to `int` types for calculations in kernels.

5.5 Priority Number Permutation

Creating a permutation of priority numbers is a requirement of the update cycle to support the multi-phase update and retain original model behaviour. There are two simple approaches to creating permutations of the priority number on the host for use on the device.

The first approach is to either use an algorithm provided by the C++ STL or write a custom method to shuffle the priority numbers then synchronously copy to the device at the beginning of each update cycle. The second approach is to execute the shuffling on the host while the simulation is updated using C++ threads and CUDA streams. This approach requires extra memory to store two copies of the priority numbers in an appropriate container. Shuffling occurs on the copy of the priority numbers that are not being used by the device.

Generating priority number on the device can provide quick permutations of the priority numbers for simulation use and free up extra device resources required by the host approach. Implementation of shuffling or permutation algorithms such as the Fisher-Yates shuffle [194] on GPU architectures is not a simple task as issues such as race conditions, optimal memory use and parallel random number generation must be considered. The generation of a parallel random permutation of priority numbers on GPUs can be implemented by extending the dart algorithm presented in the technical report by Guojing Cong and David Bader [195].

5.5.1 CPU Permutation

A simple solution to achieve random permutations of priority numbers on the CPU is to use a host function to shuffle the PN. Thrust device vectors offer iterators that are interoperable with STL algorithms. STL provides a `random_shuffle()` function which rearranges elements such that each possible permutation of the items has equal chances. There are three algorithms which can either take a user-specified random number generator or uses the default STL `rand()`. Two have been deprecated since C++14 and removed in C++17 due to issues such as the STL `rand()` not providing

uniformly distributed values. The remaining STL `shuffle()` function takes iterators and a random number generator that must meet the requirements of the*UniformRandomNumberGenerator* of the library.

A custom implementation of the Fisher-Yates algorithm shown in Listing 5.5 that takes a XORWOW random number generator to swap elements is also sufficient to provide random permutations of priority numbers.

Listing 5.5: Fisher-Yates Shuffle

```
for(unsigned int i = move_numbers.size() -1; i > 0; i--) {
    unsigned int j = ( xorwow() / ((float)4294967295)) * (i + 1);
    int k = move_numbers[i];
    move_numbers[i] = move_numbers[j];
    move_numbers[j] = k;
}
```

After a random permutation of the priority numbers is created in host memory, a synchronous copy to device memory is required for use each step. Thrust offers some general copy functions that can copy a whole array or specific range of elements. The thrust `copy_n()` function can be used to copy ranges of unique priority numbers to the device states for each species.

5.5.2 CPU A-Synchronous Permutation

The second approach to create permutations using the CPU is to perform the computation on the host while the simulation is updated on the device. This can be done using C++ STL threads that executes the host shuffle function before copying them to the device using a CUDA stream separate from the update stream. This approach is useful when the time taken to create a random permutation and copy to the device is less than the time required to update a simulation step. However, this requires the coordination of threads to protect data and extra device resources to copy from host to device.

Figure 5.9 illustrates how the C++ threads and CUDA streams can be combined to shuffle priority numbers and update the model simultaneously. The C++ condition variables and mutexes are used to keep threads in a sleep cycle while waiting for a condition to be woken. The $shuffle\_complete$ and $swapped$ boolean flags are used for synchronisation, and the simulation thread will wait at the start of the update cycle if a random shuffle has not been completed. Waiting at the start is a safety mechanism for when the update of the model is completed before shuffling. Once the simulation thread is notified to proceed this thread signals back to the shuffling thread that it is safe to begin shuffling the container with un-used priority numbers.

5.5.3 Dart Throwing Permutation

The generic dart throwing algorithm provides implicit random permutations by compacting an array of randomly placed elements. The size of an array in which darts are placed is linear in N number of elements to permute and represents a medium or dart board in which the dart is to hit. Each point is discrete in the sense that only one dart can hit that point; thus other darts arriving later fails to land and must be re-thrown to find another free spot. Figure 5.10 is an example of 8 unique integer elements or darts that are thrown to random indices of a dartboard represented as an array of sized

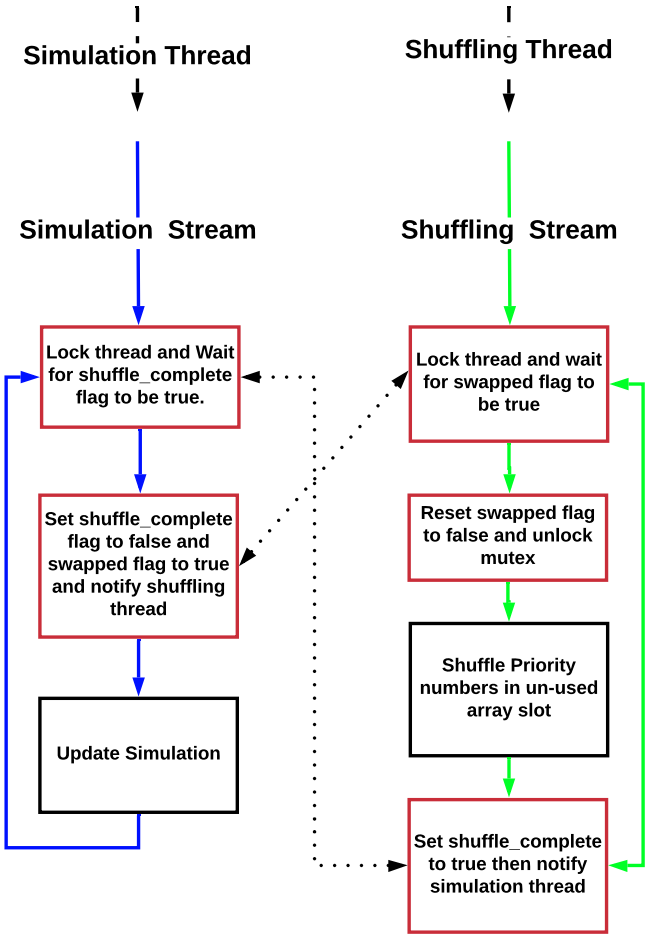

Figure 5.9: C++ threads and CUDA streams that are controlled by C++ mutex and condition variables coloured in red to coordinate shuffling concurrently with simulation update.

16. The array is compressed by removing non-occupied indices to provide a permutation of "5, 0, 2, 6, 1, 4, 3, 7".

The random placement of elements in Figure 5.10 is an ideal scenario to create a random permutation. However, it is possible that two darts may be thrown to the same point. There are two approaches of dart throwing described by Bader [195] using POSIX threads on a multi-processor. The first approach uses rounds of dart throwing where darts are thrown at random slots in a continuous region of an array and in the case of collisions those darts are re-thrown in the next round. The other approach is to use some form of mutual exclusion for darts to claim ownership of the randomly picked slot thus causing other colliding darts to be re-thrown until an empty slot is found.

The second approach of dart throwing can be implemented on the GPU to provide random priority number permutations by using atomic operations for collision resolution. The medium or dart board is represented as a device array of sized $2N$ where N is the number of priority numbers

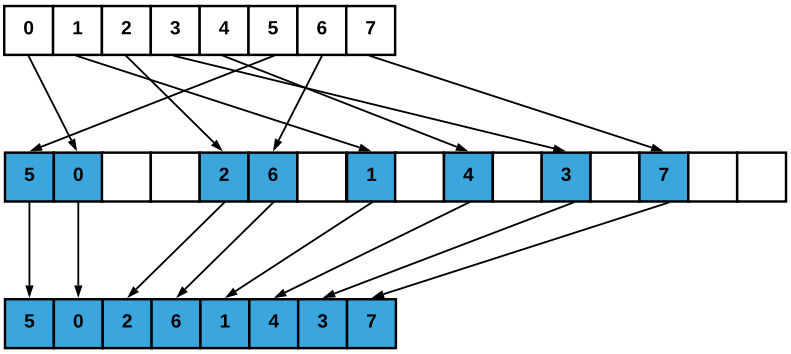

Figure 5.10: Eight darts are thrown to random slots of a 16 sized array dartboard that is compacted to remove non-hit slots to provide a permutation.

to permute. The throwing mechanism is achieved by generating an index using a random number generator in which the dart is to land in the array. The index is then used to retrieve an address to perform an atomic compare to check if the index in the array is free for the dart to be placed. The compare and swap process is an exclusive thread operation, so threads that failed to swap are required re-throw the dart at another randomly generated index.

Compaction of empty dart board slots is required if the size of the dartboard is greater than the number of darts to be thrown. The CUB library provides a device-wide select-if template function that uses a `select_op` functor to selectively copy items from one device array to another whilst retaining the original relative ordering. In this case, the `select_op` functor is specified to select placed dart board indices of the dartboard array. These placed dart indices copied into the priority number array based on its original order. The CUB select-if method matches the performance of Thrusts compaction methods on older architectures such as the Tesla C1060 and vastly outperforms on newer GPU [196].

The compacted array is a permutation of the elements which could be used for priority numbers in the Animat implementation however the compression phase is not required if the priority numbers are generated by assigning an Animat agent to each thread and use the random dart placement index as their unique priority number. This satisfies the requirements of a priority number by retaining the unique and ordered values provided by the atomic compare and swap operations. Figure 5.11 is an illustration of the compression stage being replaced by the unique and randomly allocated dart board indices as priority numbers. Although the compaction phase is not required, it is not necessary to remove it as the time taken to execute the CUB select-if is negligible relative to the time taken for an update cycle.

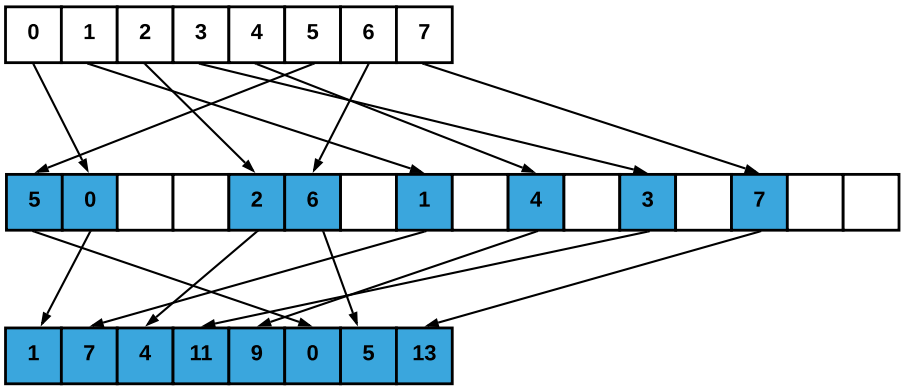

Figure 5.11: The slot index where the dart lands can be used as unique values that satisfies the requirements of a priority number.

5.6 Decide Action

The decide action CUDA kernel is implemented with the assignment of threads to each Animat agent to make use of the desired memory access patterns provided by the compressed sparse data structure. The decide action CUDA kernels are implemented for each species as individuals of a species are more likely to access the same data. Listing 5.6 is an example of a prey decide actions CUDA kernel where threads of a block are assigned to each individual. The Animat agent performs a health and age toll check before a decision is allowed to be made. The decide action Kernel iterates through prioritised rules of the rule set and in the case of a typical Animat model the priority can be breed, graze, move in a random direction, seek own species, move away from own species then move away from predators.

Listing 5.6: Prey Decide Action CUDA Kernel

```
__global__ void  prey_decide_action(DeviceState dPreyState, DeviceState dPredState,
    DeviceHelperState dHelperState, ){

   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   if(tid >= dPreyState.live_agents){ return; }

   //Reset eaten_by and decision at start of update cycle
   dPreyState.d_eaten_by_ptr[tid] = INT_MAX;
   dPreyState.d_action_ptr[tid]  = '?';

   bool alive           = dPreyState.d_alive_ptr[tid];
   //only update live agents
   if(!alive ){ return; }
   int age           = dPreyState.d_age_ptr[tid];
   . . . // Load this agent's data

   // Check if agent still alive after health and age toll of update cycle
   if(update_health_age_check(age, updated_health, alive, max_age)){
       . . . return; // Update associated variables
   }

   dPreyState.age_ptr[tid] = age;
   . . . // Load more of this agent's data

   // -1 as cell count includes self
   int prey_crowd_count = dPreyState.d_neighbour_count[position] - 1;
   int pred_crowd_count = dPredState.d_neighbour_count[position];

   // Breed Rule
   if(pre_hunger_wellfed_check(max_health, updated_health, 0 , 3 ) && prey_crowd_count > 0
       && prey_crowd_count < c_speciesData->crowd_breeding){
      int partner_index = dPreyState.d_data_A_ptr[tid];
      partner_index = -99;
      if(breed(dHelperState.d_rng_states, position, id, tid, partner_index, c_speciesData,
          dPreyState)){
         . . . return; // Update associated variables
      }
   }
   ... // Other Rules in Ruleset order
}
```

Once the condition is satisfied for the rule, the thread assigned to the individual will call the corresponding device function of the rule to determine if the decision is valid by returning a true or false boolean value. A true value means the decision is valid and can be written to the $action\_ptr$ array, and the thread returns thereafter. In the case of an invalid decision, the thread will continue to check the next rule.

An example device function used to validate the graze rule can be seen in Listing 5.7 where no grass in the current location and a failed attempt to grazed determined by a random number as a percentage returns a false value. Each of these device functions matches the action implementations of a typical Animat model discussed in Chapter 3.

Listing 5.7: Prey Decide Graze Device Function

```
__device__ bool  graze(curandState *rng_state, int position, int tid){

   int y = calcY(position);
   int x = calcX(position);
   // N% chance to graze if grass is available where N = the grazing success percentage.
   if( y <  d_grass_min_y || y >=  d_grass_max_y || x < d_grass_min_x || x >= d_grass_max_x
        || (random_int(1, 100, tid, rng_state)) > c_speciesData.grazing_success){;
      return false;
   }

   return true; // Decision to graze is valid.
}
```

The rule set implementation is fixed in the kernel example from Listing 5.6. An implementation of a dynamic rule sets using experimental device Lambdas is discussed in Chapter 7.

5.7 Conflict Resolution

Conflicting decisions can occur when two or more Animat agents decide to consume or obtain the same unique resource. In the typical Animat model, conflicting decisions can occur when two or more predators decide to eat specific prey. In order to resolve conflicts, the decision CUDA kernel can be further separated by conflicting and non-conflicting rules. The non-conflicting rules can all be decided simultaneously while conflicting rules must be iteratively solved to distribute unique resources to individuals correctly. Figure 5.12 illustrates the synchronisation requirement of rules that cause conflicts. Any rules that are prioritised before or after the conflicting decision requires synchronisation of the stream in between kernel calls.

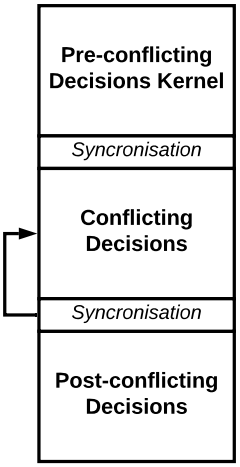

Figure 5.12: Separated and sequenced CUDA kernels to address conflicting and non-conflicting Animat agent decisions.

To describe the conflict resolution process a simple case using a grid of 9 cells with prey in the green central position surrounded by predators with their priority numbers shown in Figure 5.13. If

the surrounding predators all decide to consume the same prey in the central green cell then their decisions would cause a conflict. This conflict can be correctly resolved by using atomic operations to compare the priority numbers of each individual to determine which predator is allowed to consume the prey.

Any predator that is unsuccessful in hunting a prey because another predator with a lower priority number hunts it first will never need to re-examine the prey as the other predator will only relinquish it to another even higher priority predator. The predator can only continue to the next rule if no other prey is around to examine.

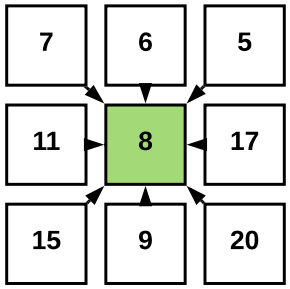

Figure 5.13: Example of predators and their priority numbers in the eight surrounding cells deciding to consume the prey in the green central cell.

If the priority numbers are ordered to replicate the original behaviour, the lowest value can be used to represent a higher priority predator. The CUDA `atomicMin()` function can be used to efficiently determine the lowest value priority number as the atomic function does not act as a memory fence and thus does not imply synchronisation or ordering constraints. If the lower value priority number means higher priority, the agent with priority number 5 is allowed to consume the central prey as seen in Figure 5.13. Alternatively, if a higher value priority number system is used then the `atomicMax()` function can be used.

During an update cycle of a typical Animat model, conflicts can potentially arise anywhere predators are within the interaction distance of prey agents. Since thread launches are managed by CUDA there is no guarantee for the order of `atomicMin()` comparison of priority numbers. This situation can be solved by calling a conflict resolution kernel multiple times until none of the predators detect any conflicts.

The conflict resolution kernel contains a shared conflict flag and a conflict resolution device function. This device function is of a boolean type and will return a `true` if there was a conflict or `false` if not. These boolean values can be used to set the shared conflict flag for each iteration.

5.7.1 Iterative Conflict Resolution

The conflict resolution kernel is iteratively executed and synchronised until all conflicting decisions are resolved by using the device function in Listing 5.8. A conflict is signalled using a shared conflict flag of the boolean type. The shared conflict flag does not require protection from any race conditions as Animat agents can only set the flag to true thus a conflict raised by one or more agents results in

another synchronised round of conflict resolution.

Listing 5.8: Predator Decide Eat Conflict Resolution Device Function

```
__device__ bool  pred_decide_eat_conflicts(int position, int id, int move_number, int &
    eating_index, DeviceState dPreyState,SpeciesData *speciesData, int *prey_start, int
    move_number_vec_idx){

   for(int i = 0; i < 9; ++i){
      // Offsetted neighbouring indices to check for
      int c_y = (int)c_y_offsets_ptr[i];
      int c_x = (int)c_x_offsets_ptr[i];
      int look_at_yx = calc_location(position, c_y, c_x);

      int s = prey_start[look_at_yx];
      if(s < 0) continue;
      for(int j = s; j < dPreyState.live_agents; ++j){
         int target_position = dPreyState.d_location_ptr[j];
         int target_move_number = dPreyState.d_move_number_ptr[move_number_vec_idx][j];
         bool target_alive = dPreyState.d_alive_ptr[j];

         // check if look up gone too far
         if( target_position > look_at_yx) { break; }

         // If prey went first and died then the predator can't eat this prey
         if( (move_number > target_move_number) && (target_alive == false)){
            continue;
         }

         int min_eaten_by = dPreyState.d_eaten_by_ptr[j];
         if(min_eaten_by < move_number) {continue; }

         // Atommically returns the min and writes the lower value to old *address
         min_eaten_by = atomicMin(&dPreyState.d_eaten_by_ptr[j], move_number);

         // Prey is already marked for consumption by another higher ranking pred
         if(min_eaten_by < move_number){
            continue;
         } // Take marked prey from low ranking predator
         else if(min_eaten_by != INT_MAX && min_eaten_by != move_number){
            eating_index = j;
            return true;
         } // Found a prey to eat without causing conflicts
         else {
            eating_index = j;
            return false;
         }
      }
   }
   eating_index = INT_MAX;
   return false;
}
```

Listing 5.8 is the device function used to resolve and signal predator eat conflicts. The predator begins the search by looking in each Moore's neighbourhood cell, and each cell may contain multiple preys that are checked if they can be eaten.

Once a potential prey is found the predator requires data from each prey such as the priority number, status, and eaten by for comparison. Prey agents with higher priority numbers can be a potential food source regardless of their dead or alive status. This is due to the sequence of the original model in which the prey is scheduled for a later update.

The last part of the algorithm is the crucial section that determines if a predator has caused a

conflict while marking the prey for consumption. A `atomicMin` CUDA function is used to compare and write the predators priority number with the preys eaten by value. The result is written back to the same memory address of the eaten by variable, and the function returns the lowest value. In Listing 5.8 the value returned from the atomic minimum function is stored in a local variable min_eaten_by and will eventuate to any of the three outcomes.

1. The min_eaten_by is a lower value than the predators priority number thus the prey is a potential food source for another predator, so the predator continues the search for other prey.
2. The priority number of the predator is smaller than the min_eaten_by value, indicating the prey is available to be a potential food source whilst bumping of another predator with lower priority causing a conflict.
3. The prey is available to be a potential food source with no conflicts arising; this occurs when min_eaten_by is either equal to the predators priority number or the maximum value of the primitive type used to represent the priority number.

The min_eaten_by of the prey is set to the maximum value of the data type used at the start of each update cycle. The min_eaten_by value either remains the same as the initial value when the prey is not marked for consumption or settles on the priority number of the predator that decides to consume it.

Using the simple case shown in Figure 5.13 the surrounding predators simultaneously search their neighbourhood and find the prey in green with priority number 8 then loads the required information of the prey such as its priority number and state as seen in lines 3 to 14 of Listing 5.8. All the predators then compare their priority number with the prey to check if that prey with a lower priority number has died when if it was to be updated first as shown in lines 16 to 22. And in this case only predators with priority numbers 5, 6 and 7 may progress to compare their move numbers to see who can eat the prey using an atomic operation as seen in lines 24 to 41.

5.8 Perform Actions

The decisions of each Animat agent are processed once all individuals have decided on a rule they are permitted to perform. These actions can be grouped based on the rule requirements as discussed in Section 5.4. Interactive rules such as eating or breeding require the knowledge of the other individuals decision to perform the action correctly. Non-interactive rules such as grazing or moving only require the state of their individual environment which does not change until the end of the update cycle. The grazing rule may belong to the interactive action group if the environment of the model is configured with limited grass that needs to regrow when grazed.

Similar to the decide action CUDA kernels, Animat agents are processed in an order based on their species. Updating individuals in this sequence may help to avoid processing of Animat agents that will die of circumstances such as predation. Figure 5.14 illustrates this order in which predators perform their actions first; thus any prey that has been eaten is not required to be updated. Although threads are assigned to eaten prey of an unsorted container which results in divergence

and inactivity. It is more efficient than reorganising the containers and associated data buffers such as $data\_A$ which may contain the index to other Animat agents related to an individuals decision. This is a relatively small part of the computational costs.

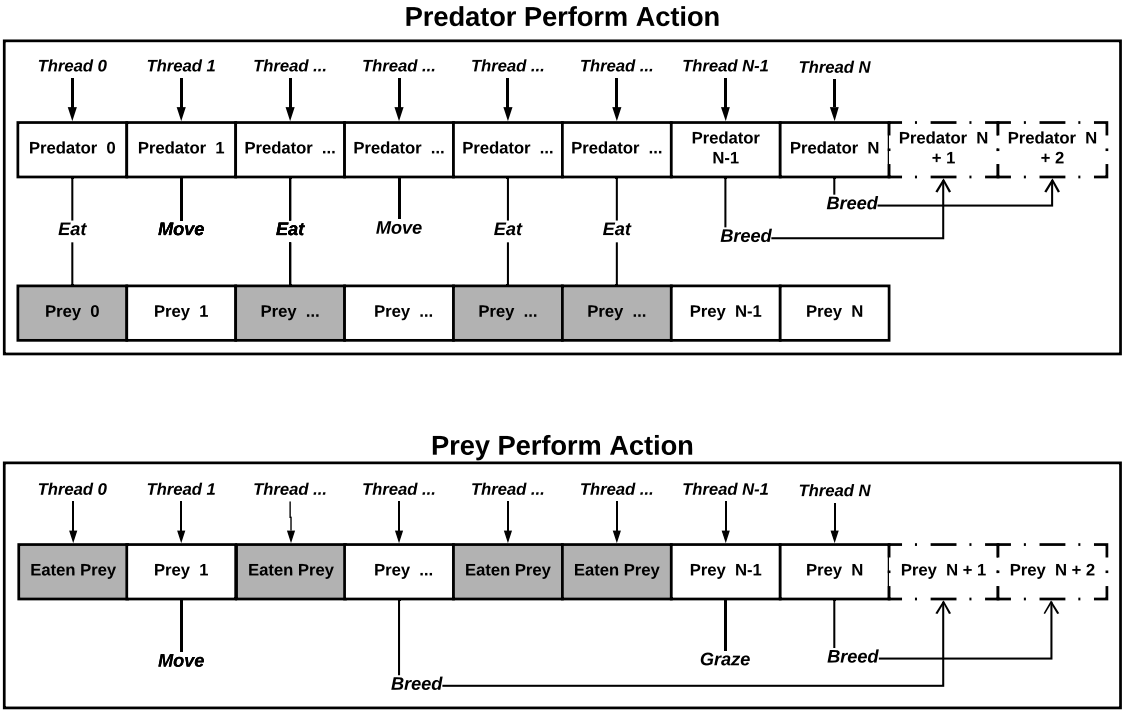

Figure 5.14: Predator performs actions before prey: eaten prey marked in grey are not required to be updated.

Predators preforming the eat action must know the priority number of the prey it decides to consume as prey that is prioritised to be updated first may have decided to perform a grazing action. In such cases, the predator must consider the health of the prey after it has grazed so that it consumes the correct amount of nutrients. The breeding action of both species may also depend on the eating or grazing action of the partnering individuals as newly bred agents may take the average health value of their parents.

5.8.1 Predator Perform Eat

The perform eat CUDA kernel for predators is stream synchronised to apply health updates before breeding can occur. To reproduce the eating action in the original model, the predators must compare their priority numbers to the prey intended for consumption to calculate the correct nutrient intake. This process is shown in Listing 5.9 in which there are three possible scenarios.

The first scenario is when the predator has priority which means the predator's new health is calculated by adding the result of $(health - health\_toll) + (prey\_health \times nutrient\_intake\_rate)$ that is capped at the species maximum health.

The second scenario is when prey has priority of update and decides to graze. This scenario requires the predator to obtain the nutrients of the prey after it has grazed which is calculated as $target\_health = (prey\_health - health\_toll + grazed\_nutrients) \times 1.5$. The $target\_health$ must be capped at the species maximum health.

The last scenario is when prey has priority but does not decide to graze. In this case the $target\_health$ can be calculated as $target\_health = (prey\_health - health\_toll). \times 1.5$.

Listing 5.9: Predator Perform Eat

```
//update the new health values
__global__ void  predator_perform_eat(DeviceVecPtrSize preysList, DeviceVecPtrSize
    predatorsList, SpeciesData *speciesData, int *prey_start, int *pred_start, curandState
    *rng_state , int move_number_vec_idx){

   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if(tid >= predatorsList.live_agents){ return; }

   bool alive           = predatorsList.alive_ptr[tid];
   char action          = predatorsList.action_ptr[tid];

   // If agent is still alive and found a prey to eat
   if(!alive || action != 'E'){ return; }

   int eating_index     = predatorsList.eaten_by_ptr[tid];
   int move_number         = predatorsList.move_number_ptr[move_number_vec_idx][tid];
   int target_move_number  = preysList.move_number_ptr[move_number_vec_idx][eating_index];
   char target_action      = preysList.action_ptr[eating_index];
   int target_health       = preysList.health_ptr[eating_index];
   int health           = predatorsList.health_ptr[tid];
   int updated_health      = health - 10;

   // If predator has higher priority
   if(move_number < target_move_number){
      updated_health += (int)(target_health * 1.5);
      if(updated_health > 200){
         updated_health = 200;
      }
      preysList.alive_ptr[eating_index] = false;
      predatorsList.data_A_ptr[tid]      = updated_health;
   }// Consuming prey with high priority that could have grazed
   else if(move_number > target_move_number){
      if(target_action == 'g'){
         int target_grazed_health = target_health - 10 + grass_nutrient;
         if(target_grazed_health < 100){
            target_grazed_health = 100;
         }
         target_health = target_grazed_health;
      }
      else{ //if Prey went first and not grazed get updated health of Prey
         target_health -= 10;
      }
      updated_health += (int)(target_health * 1.5);
      if(updated_health > 200){
         updated_health = 200;
      }
      preysList.alive_ptr[eating_index] = false;
      predatorsList.data_A_ptr[tid]      = updated_health;
   }
}
```

5.8.2 Agent Perform Breed

Animat offsprings are added to the end of the main container using a shared index variable that functions like an accumulating semaphore which is reset to the number of live agents at the start of each step. The atomic add function allows Animat agents to perform breed actions simultaneously while retrieving unique indices to place the offspring. There is no requirement for the order of inserting newly bred agents.

Newly bred agents require three initialisation values such as the health, location and identification. The initial health value is an average of the parent's health. The Animat agent performing the breed function must consider the partners priority number to determine the correct health value to average. Partners may have update priority and have decided to eat or graze. In this case, the contribution of the partner's health nutrients is calculated as $partner\_health = (partner\_current\_health - health\_toll + consumed\_nutrients)$ which is capped to the species maximum health. If the partner has priority but does not decide to eat or graze then $partner\_health = (partner\_current\_health - health\_toll)$.

The newly bred agent can be inserted in the same environment location as the parent performing the breed action. Identification is a value that is required to be unique, and the `atomicAdd` operation can be used on a shared *unique_id* variable to assign unique id's.

5.9 Post-Processing

At the end of each update cycle, the compressed sparse data structure is required to be re-arranged to the correct state for the next update cycle. This process involves removing dead Animat agents, adding new agents, sorting the agents by their new position and resetting any associated indexing containers.

5.9.1 Sorting

The CUB or Thrust library can be used to sort Animat agents by their location in the environment. Thrust sorting actually uses CUBs block-wide and device-wide sorting methods in their underlying implementation to achieve high performance. Thrust sorting is implemented with algorithms that are library managed based on the arguments provided to the function. The high performing parallel radix sorting algorithm is used if Thrust can determine that the data type to sort is an inbuilt numeric type such as `int`, and the comparison function is the `thrust::less<int>`. To achieve better performing radix sorts on Thrust, it is possible to use primitive data types with enough bits such as a 16 bit `short` to contain a grid size of 256 by 256 offering 65536 k-indices. Thrust can dynamically exploit unused key bits to reduce the cost of sorting with only 4 bits required to be re-arranged for a 16-bit `short int`.

A benefit of using Thrust for sorting is the libraries fancy zip iterator that takes a tuple with a virtual range to sort all the associated containers by a key. The tuple is created by providing the `begin()` and `end()` iterators of the data containers that is to be sorted. Listing 5.10 is an example

of using the zip iterator to sort Animat agents by their current position which is stored as a k-index in the *location device_vectors*.

Listing 5.10: Sort Agents by K-index

```
void  sort_agents_yx(DeviceAgents &agents, cudaStream_t *stream, int free_indices){
   thrust::sort_by_key(thrust::cuda::par.on(*stream),
      agents.location.begin(),
      agents.location.end(),
      thrust::make_zip_iterator(thrust::make_tuple(
         agents.data_A.begin(), agents.id.begin(),
         agents.move_number[free_indices].begin(),
         agents.age.begin(), agents.health.begin(),
         agents.alive.begin())
      ),
      thrust::less<int>()
   );
}
```

Listing 5.11: Sort, Gather then Copy

```
void  sort_agents_yx(DeviceAgents &agents, cudaStream_t *stream, int free_indices){
   // Number of agents
   unsigned int N = agents.location.end()-agents.location.begin();

   if(tmp_int == nullptr) { tmp_int = new thrust::device_vector<int>(N); }
   if(tmp_bool == nullptr) { tmp_bool = new thrust::device_vector<bool>(N); }
   if(map == nullptr) { map = new thrust::device_vector<unsigned int>(N); }

   tmp_int->reserve(N);
   tmp_bool->reserve(N);
   map->reserve(N);

   // Generate Sequence Map
   thrust::sequence(thrust::cuda::par.on(*stream), map->begin(), map->begin()+N);

   // Sort Map
   thrust::sort_by_key(thrust::cuda::par.on(*stream), agents.location.begin(), agents.
       location.end(), map->begin(),thrust::less<int>());

   // Permute Data_A
   thrust::gather(thrust::cuda::par.on(*stream), map->begin(), map->begin()+N, agents.
       data_A.begin(), tmp_int->begin());
   thrust::copy(thrust::cuda::par.on(*stream), tmp_int->begin(), tmp_int->begin()+N, agents
       .data_A.begin());

   // Permute Priority Number
   thrust::gather(thrust::cuda::par.on(*stream), map->begin(), map->begin()+N, agents.
       move_number[free_indices].begin(), tmp_int->begin());
   thrust::copy(thrust::cuda::par.on(*stream), tmp_int->begin(), tmp_int->begin()+N, agents
       .move_number[free_indices].begin());

   ... // Permute all other Agent attribute
}
```

Although the zip iterator is a quick and easy way to implement sorting for the compressed sparse structure, there are performance issues when using the functionality to sort multiple containers. Thrust manages the memory allocation of temporary containers used for sorting and may reallocate device memory for each function call which can be costly if used iteratively to sort multiple containers. An alternate approach is to use the radix sort on a single temporary container representing the key, which is used to provide a sequence to re-arrange the associated containers.

The containers are permuted using the gather and copy functions that are provided with the temporary containers as the parameters. This approach enables the use of dynamic device memory for temporary containers which can be reused by appropriately resizing the storage to accommodate the dynamic requirements. Listing 5.11 shows the implementation of the sort, gather then copy alternative which provides better performance and also requires less device memory.

5.9.2 Reset Containers

Resetting the start index array of the compressed sparse structure and pre-processed storage is a requirement to prepare the storage for the next update cycle, pre-processing data is described in Section 5.10. The default values can be initialised and stored in device global memory for low-cost device to device memory copies, effectively resetting the pre-processed arrays.

5.10 Pre-Processing

Sharing data is a common way to improve the performance of CUDA applications. CUDA threads achieve this using shared memory in a block or shuffle instructions on a warp in modern architectures. In the Animat CUDA implementation, the data access patterns are investigated to determine which data can be shared or reused each time step. The following is a list of data in a typical Animat model that can be pre-processed to be re-used during the update cycle.

- Count of neighbouring Animat agents.
- Index of first and last Animat agents in a neighbourhood.
- Index of first Animat agent in vision for each species.

5.10.1 Neighbouring Animat Cell Count

Re-using data in an update cycle refers to the multiple Animat agents occupying the same cell location requiring the same pre-calculated information. This can be useful as it is probable that there are numerous Animat agents in the same cell when the standard density of the model has been reached.

Breeding and grazing are high priority actions for the survival of individuals, and there are crowding rules of the model to force Animat agents to move when the neighbourhood of a cell location becomes overcrowded. The pre-count of neighbouring Animat agents can be performed once for each cell at the start of each update cycle and can be re-used by Animat agents during the cycle.

A CUDA kernel to count neighbours can be implemented as shown in Listing 5.12 in which threads load cell counts into shared memory for each thread mapped to an occupied cell to load into thread local memory to tally the neighbour counts and find the index of the first and last agent in the Moore neighbourhood. Breeding is an example decision that requires the search of the last partnering Animat agent within the Moore neighbourhood which is based on the configuration of a typical Animat model.

Listing 5.12: Count neighbours and find the index of first and last agents in a neighbourhood.

```
__global__ void count_neighbours_set_first_last(DeviceState state, DeviceState other_state)
    {

   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;

   if(x >= d_grid_width || y >= d_grid_height) { return; }

   int ym1 = y-1;
   int yp1 = y+1;
   int xm1 = x-1;
   int xp1 = x+1;

   __shared__ int s_agent_cell_count[34*10]; // Configure Shared memory size

   const int shared_width = 34; // Count own cell and 8 neighbours

   // Store in local memory
   int pos_x[9];
   int cell_x[9];

   //for setting position of first and last
   pos_x[0] = y * d_grid_width + x;
   pos_x[1] = y * d_grid_width + xm1;
   pos_x[2] = ym1 * d_grid_width + x;
   pos_x[3] = yp1 * d_grid_width + x;
   ... // Compute Other 4 to 8

   int bx = threadIdx.x;
   int by = threadIdx.y;

   // Load into shared memory own cell
   cell_x[0] = state.d_cell_count[pos_x[0]];
   s_agent_cell_count[(by+1) * shared_width + bx + 1] = cell_x[0];

   if((by * blockDim.x + bx) < 32) { // Thread 0 - 31 for loading top
      if(blockIdx.y > 0) { // Not top row of grid load into top row of shared memory + 1
          column
         s_agent_cell_count[bx+1] = state.d_cell_count[pos_x[2]];
      } else { // If top row of grid nothing to load set zero
         s_agent_cell_count[bx+1] = 0;
      }
   } else if((by * blockDim.x + bx) < 64) { // Thread 32 - 63 for loading bot
      if(blockIdx.y < gridDim.y-1) { // If not bottom row of grid then load in to last row
          of shared memory + 1 column from first row of block below
         s_agent_cell_count[9 * shared_width + bx + 1] = state.d_cell_count[(blockIdx.y+1)
             * blockDim.y * d_grid_width + (blockIdx.x * blockDim.x + threadIdx.x)];
      } else { // If bottom of grid nothing to load set zero
         s_agent_cell_count[9 * shared_width + bx + 1] = 0;
      }
   }
   ... // Load left and bottom
   __syncthreads();
   // Thread finished loading to shared memory, return if no agents occuyping this cell
   if(cell_x[0] == 0 && other_state.d_cell_count[pos_x[0]] == 0){ return; }
   // Load into local memory
   cell_x[5] = s_agent_cell_count[(by)*shared_width   + bx];
   cell_x[2] = s_agent_cell_count[(by)*shared_width   + bx+1];
   cell_x[7] = s_agent_cell_count[(by)*shared_width   + bx+2];
   ... // Load mid and bottom row
   ... // Iterate each cell_x[] and accumulate neighbour count and track first and last
       agent
}
```

5.11 Summary

This chapter discusses the model and implementation requirements to simulate a typical Animat model on GPU architectures using the CUDA parallel programming platform. The CUDA platform and high performing interoperable parallel template libraries are introduced to assist the discussion of the implementation details. Model and implementation requirements include a data structure to support the high performing execution model of GPU architectures, a priority number system to allow reproducibility of the original model behaviour and a process to resolve conflicting decisions that occur. The decide and act process is a solution to update Animat agents concurrently using the CUDA programming model.

6 Results

The performance results of the CPU, Multi-Core (MT-CPU) and GPU implementations of a typical predator-prey Animat model presented in Chapter 2 have been collected for comparison. Implementation details of the Animat model on the architectures used in this thesis are discussed in separate Chapters with the CPU in Chapter 3, MT-CPU in Chapter 4 and GPU in Chapter 5.

The computational cost of each implementation is presented to show the performance of the Animat model simulation on various architectures used in this thesis. Aspects of the model such as system sizes and phases of the system population dynamics have been used to collect the performance results of certain parts of the implementations for comparison. Components of the implementation include the requirements of randomising order of Animat agents, updating the agents in the permuted order and updating the data containers as a result of processed agents.

The process to allow replication of serial model behaviour is discussed and the performance results of the iterative conflict resolution implementation are presented to show that replication of agent-based models such as the Animat can be achieved without sacrificing performance. The ranged search pattern extension to the model is used to implement and present the robustness of the GPU implementation to allow modification.

6.1 Simulation Performance

The first performance results compared is the average processing time required for one update cycle of a typical Animat simulation after the growth phase for each system size in terms of environment dimensions on four computational architectures. The processing units and memory used to execute and collect simulation data for comparison are as follows:

1. 3.50GHz base frequency Intel i7-4770K with 8MB of Cache and 32GBs of DDR3 2400 RAM at 1333MHz clock speed.

2. 2 x 16 Cores 2.2 GHZ AMD Opteron Processor 6274 with 16MB of Cache and 64GBs of DDR3 PC3-10600 at 1333MHz clock speed.

3. Tesla K20X with GB GDDR5 memory and 2688 CUDA cores on a 2.5HGhz Intel Xeon CPU E5-2640 Host.

4. RTX 2080Ti with 11GB of GDDR6 memory and 4352 CUDA cores on a 2.5HGhz Intel Xeon CPU E5-2640 Host.

The implementations of the Animat model are compiled with the g++ compiler for the CPU and CUDA NVCC for the GPU using Toolkit 10.0 on the Ubuntu 18.04 operating system. Both CPU and GPU programs use the -O3 optimisation flag during compilation and the native compute capabilities of the GPUs are used, with the K20X on 3.5 and 2080Ti on 7.5. The K20X is used as a comparison to the i7 as they are produced around the same era whereas the 2080Ti is compared to the K20X to present the performance increase that is available from new hardware without changes to the code. The AMD 1 Core performance results for system sizes seen in Figure 6.2 uses the single-core implementation as a reference with no further performance data presented as the AMD Opteron cores are designed for parallelism using lower frequency clock speeds.

A typical model configuration is used to collect performance results, and the configuration is discussed in Chapter 2 and 3. The timing data collected is after the growth phase of the model, at which the Animat agents have propagated throughout their environment to reach standard density. Population boom-bust oscillation dynamics are present throughout the various phases of a simulation run; however, they settle by oscillating around an average number after the growth phase.

6.1.1 System Size

Figure 6.2 shows the average time taken in milliseconds for an update cycle of the simulation on typical system sizes ranging from 200^2 to 1500^2 tested on a single core of an i7 CPU, Multi-AMD CPU, K20X GPU and 2080Ti GPU. The scaling of a single-core CPU performance is as expected with each larger system size four times greater than the previous. The system sizes start at 200^2 as smaller environments may result in a high chance of population crashes using a typical Animat model configuration. Figure 6.1 shows the simulation screen capture of the Animat model during standard density after the growth phase for various system sizes.

The average time to update a cycle of the simulation for each system size on a serial implementation of the model using an i7 CPU is approximately 1.5 times faster than a single core of the AMD Opteron running the same implementation. This is expected due to the differences in computational power per core for the processors, the Opteron is designed for work stations and servers with more cores than the i7 but operates at a lower frequency. For two or more cores the MT-CPU implementation of the Animat model is used for testing, and when more cores are available, there is an improvement in performance. Full utilisation of the two cores begins on a 300^2 system size and for the other N number of cores tested: 400^2 for 4 cores, 600^2 for 8 cores, and at 1200^2 for 16 cores. The 32 cores is a configuration of 2 x 16 core AMD Opteron CPUs and do not show much improvement over the 16 cores due to issues such as the resource contention of the processors.

The slight variation in the performance scaling of the 2, 4 and 8 cores is caused by many factors of the MT-CPU implementation such as the set size of the spatial domain decomposition (SDD)

blocks being 61^2 to use the smallest dimension that is just larger than the vision radius of the predator species which is 60. The configuration of SDD Block dimensions are discussed in Chapter 4, and as system sizes increase by 100^2 the required number of blocks for each tested system size does not scale proportionately. For example, the required amount of SDD Blocks of size 61^2 for 200^2 is 11, 300^2 is 25, 400^2 is 43 and 500^2 is 70 etc. Other factors include the scheduling of non-nearest neighbouring blocks and hardware resource contention. A closer observation of the time spent scheduling versus updating the model is shown in section 6.3.

The performance result scaling of the K20X and 2080Ti GPU shows that at 1500^2 the hardware is not yet fully utilised, effortlessly processing the smaller system sizes. This is due to the many cores of the GPU that have not been used as processing an Animat agent is relatively low cost when using the compressed sparse structure and associated direct indexing system to allow for coalesced memory transactions. The timing for an average cost of updating an Animat agent is presented in Section 6.2.

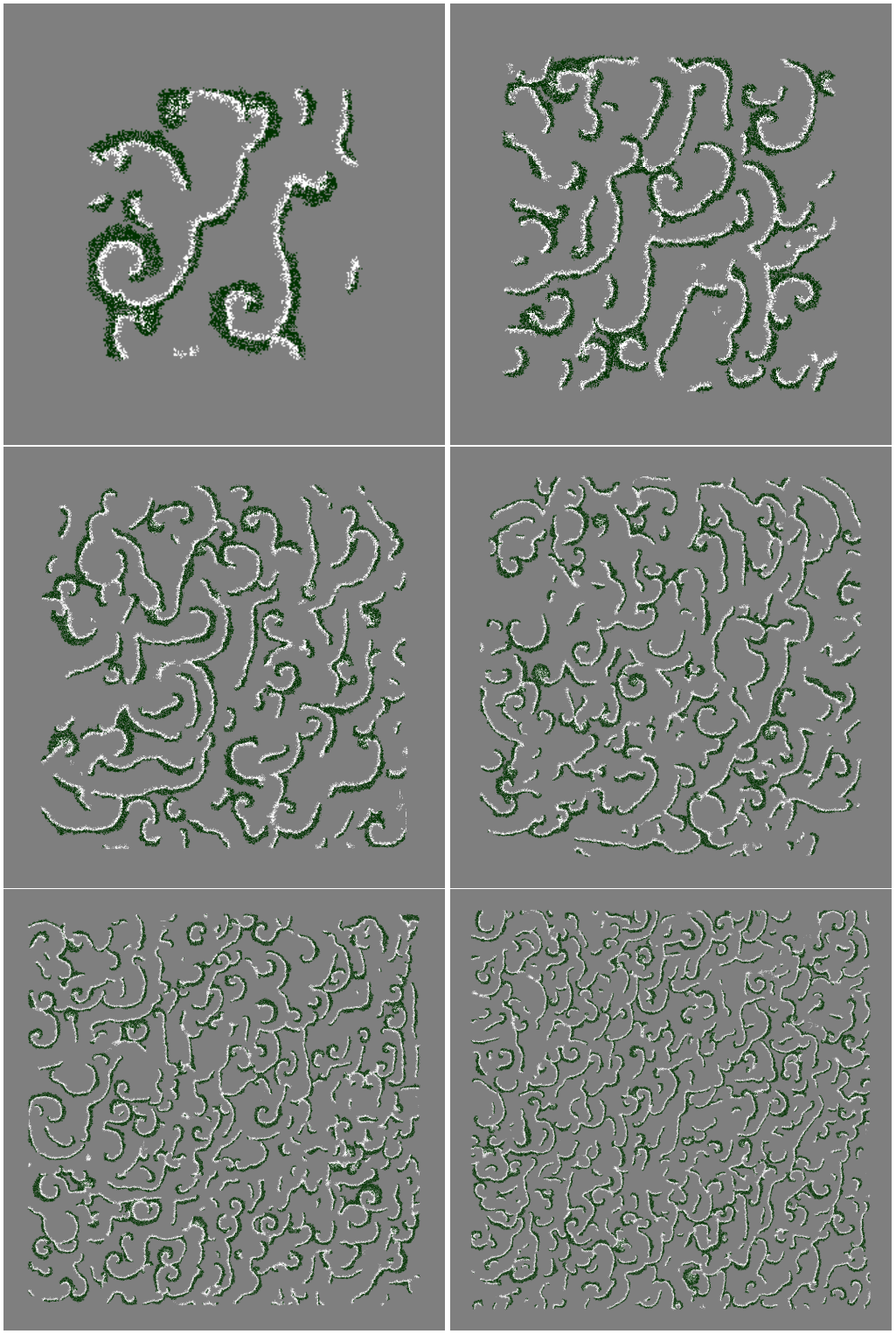

Figure 6.1: Screen capture of an Animat simulation on system sizes 200^2, 400^2, 600^2, 800^2, 1000^2, 1200^2 during model carrying capacity. Predators are coloured white and prey are coloured green.

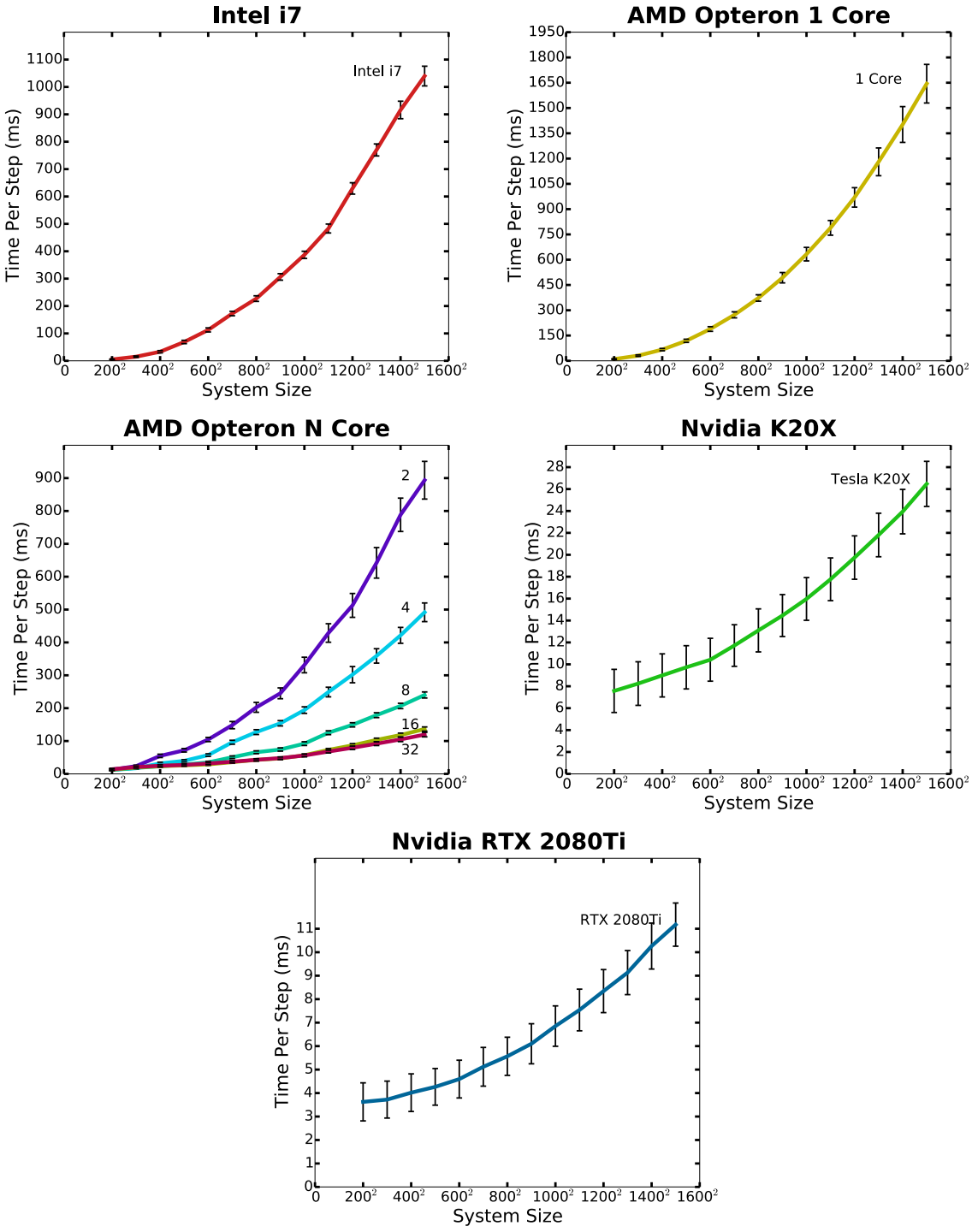

Figure 6.2: Time in milliseconds per step for system sizes 200^2 to 1500^2 on an Intel i7 CPU, 1-Core AMD CPU, N-Core AMD CPU, K20X GPU and RTX 2080Ti GPU.

6.1.2 Speed-Up

The speedup of the multi-core AMD over the single-core i7 CPU implementation is shown in Figure 6.3, 2 Cores do not provide much improvement and can be worse when the system size is smaller than 600^2. On 500^2, more cores can help to process the larger population of Animat agents. The speedup is calculated by dividing the mean time per step of each architecture by the mean time on the single core i7.

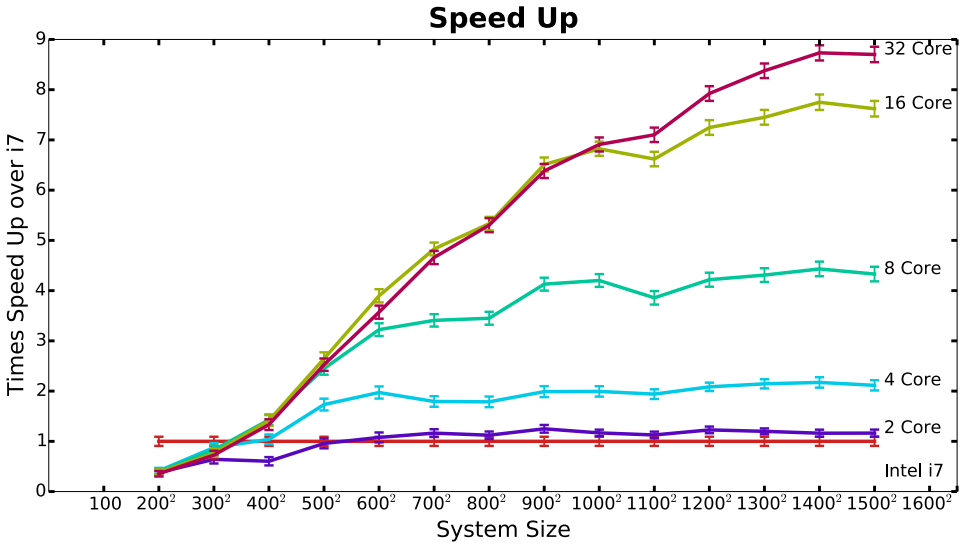

Figure 6.3: Speed up of AMD Opteron N-cores over Intel i7 for system sizes 200^2 to 1500^2. The error bars represent the standard error of the mean for each architecture on each system size that is tested.

The speedup of the GPU implementations can be seen in Figure 6.4. It may not be worth using the K20X for a system size of 200^2 however at 300^2, the K20X begins to provide performance increases and outperforms the multi-core AMD at 300^2. The K20X reaches on average a 40-times speedup over an i7 at a system size of 1500^2. The 2080Ti uses the same code as the K20X and achieves a 93 speed up approximately doubling the speedup of a K20X. Figure 6.5 shows the impressive scaling capabilities of the GPU implementation at larger system sizes ranging from 2000^2 to 5000^2. These results show that the GPU architectures can be effectively used to accelerate the Animat model simulation to larger scales that may not be feasibly simulated using single or multi-core CPUs tested in this thesis. This provides the opportunity for domain experts to explore larger system sizes that were not considered due to the expected time required.

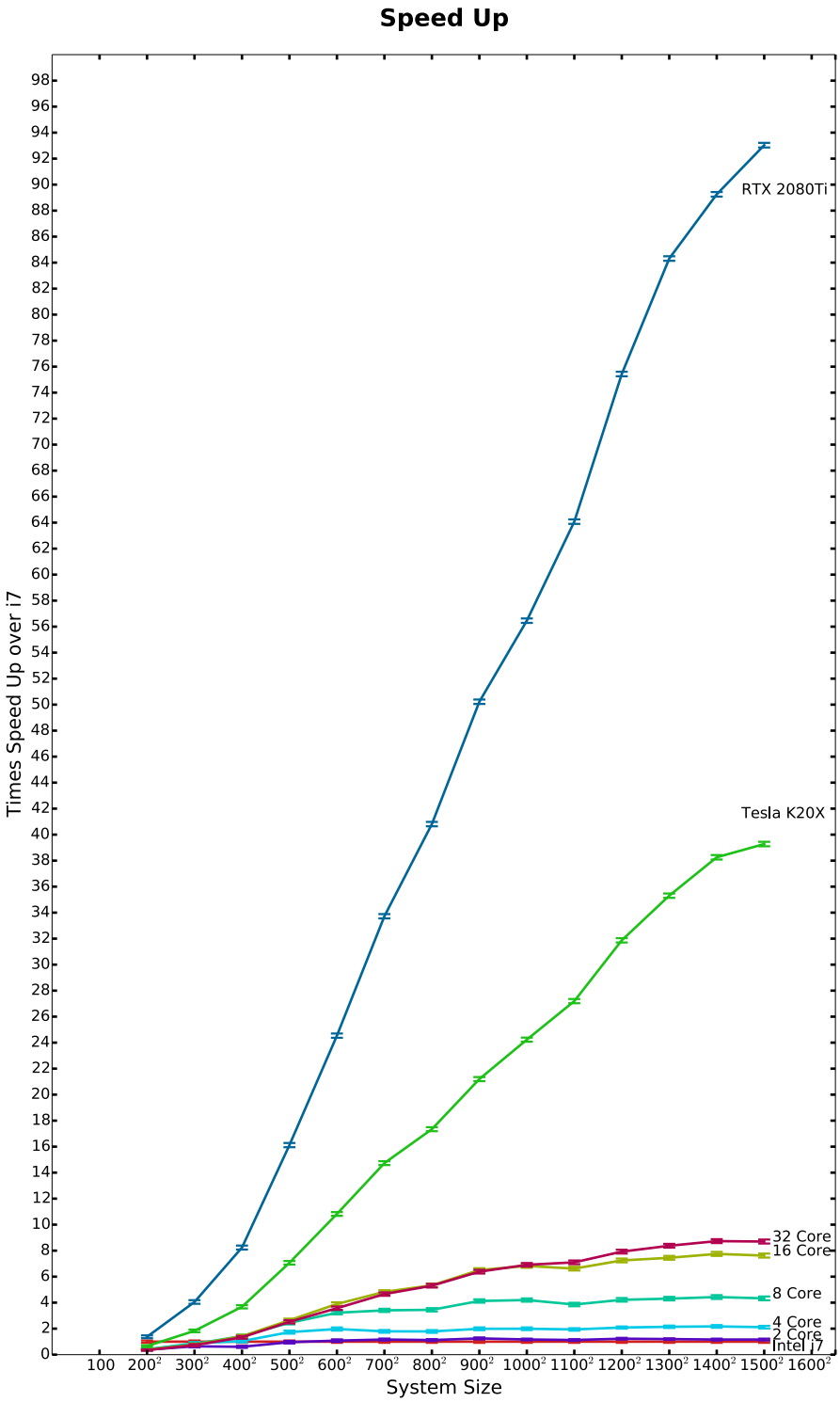

Figure 6.4: Speed up of K20X, 2080Ti and AMD N-cores over Intel i7 for system sizes 200^2 to 1500^2. The error bars represent the standard error of the mean for each architecture on each system size that is tested.

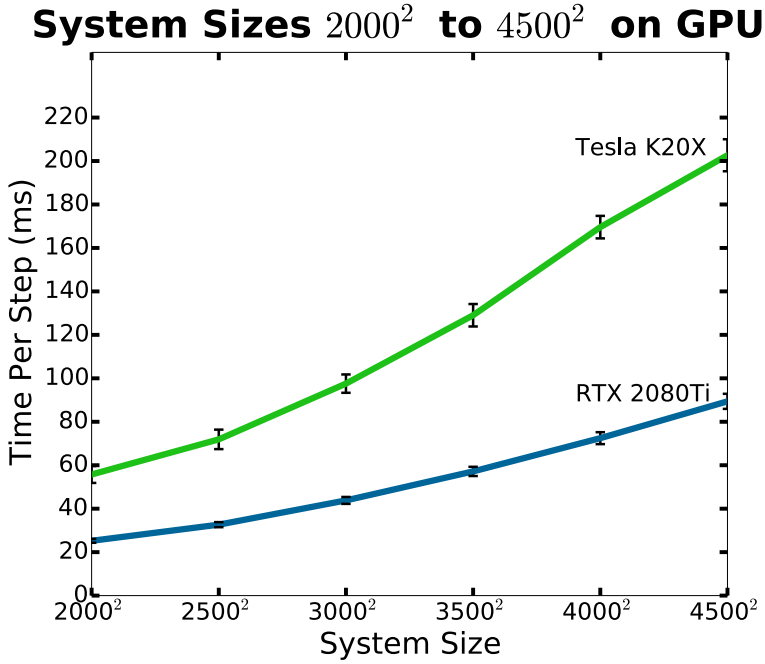

Figure 6.5: Time in milliseconds per step for larger system sizes from 2000^2 to 4500^2 on K20X and 2080Ti, this shows the scale-up. However, scale-out using multi GPUs is beyond the scope of this thesis.

6.2 Computational Costs for Agents

The performance comparison for the tests in this section are generated from the data collected to show the computational costs of updating agents per cycle. The architectures used for testing are the i7 CPU, Opteron CPU, K20X GPU and 2080Ti GPU. The data collected include the time taken to update a N number of Animat agents and the average time to process an agent of the predator and prey species.

The performance comparison for the test shown in Figure 6.6, shows a plot of times in milliseconds taken per step to update a total number of agents ranging from 1000 to 800,000. The test configuration measures the time during the growth stage of the Animat simulation for a system size of 1500^2 where a total of 1000 Animat agents are initialised in random locations within a centred, 20 cell radius circular area at which they proceed to propagate and expand to cover the full environment size to reach population carrying capacity.

The results show a linear scaling of time taken as the population numbers grow. The fluctuation in processing time is caused by the dynamics and implementation of the model on the architectures. Model dynamics that may factor in varying the range of plots include the boom-bust population cycles and the changing range of searching required by the movement of Agent clusters. Implementation factors include resource contention and conflict resolution requirements that are discussed

further in this section.

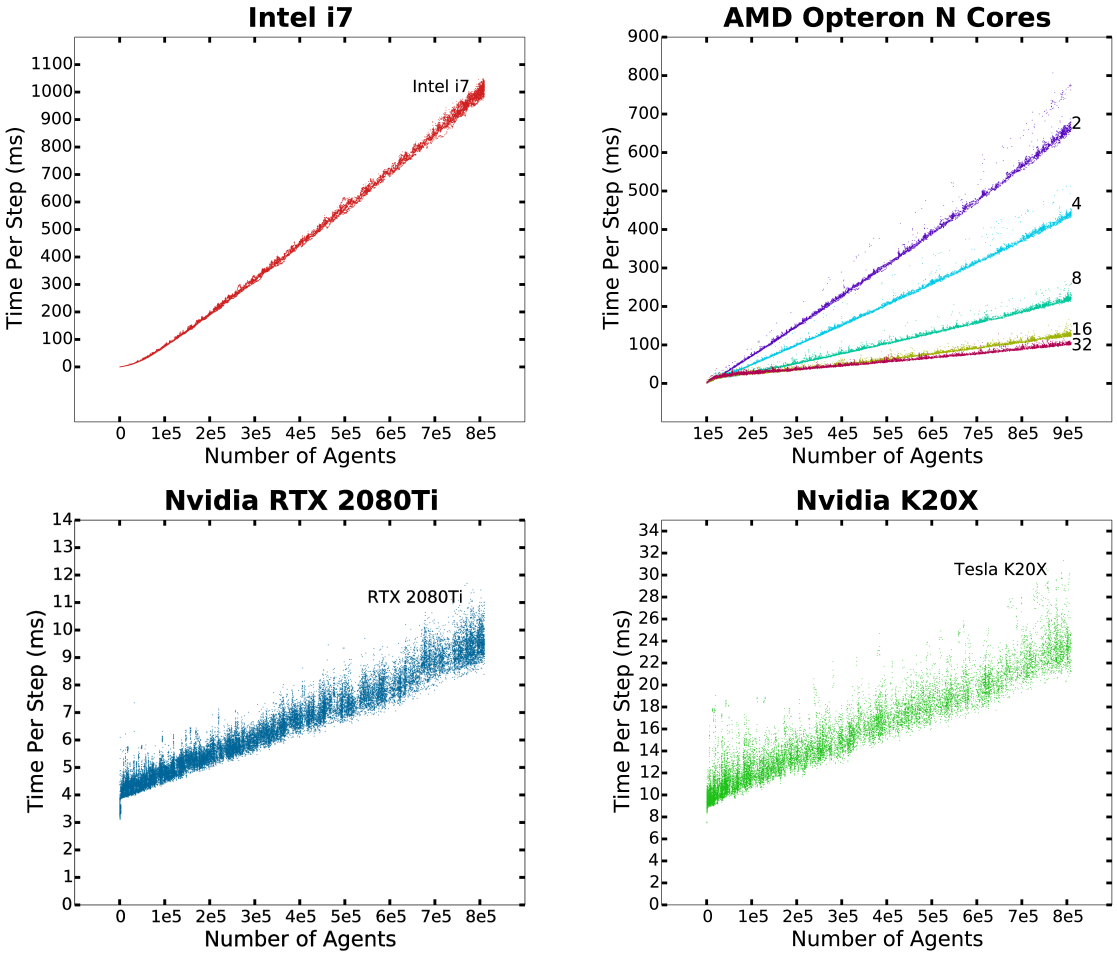

Figure 6.6: Time in milliseconds per step to update N number of agents during the growth phase on the i7, multi-core AMD, K20x and 2080Ti. The growth phase starts with 3000 Animat agents that grow their populations as they spread through the environment.

Figure 6.7 shows the plot of average time in microseconds to update each agent of the prey and predators species. The simulations start with 2000 prey and 1000 predators in random locations of a 1500^2 environment size. The large initial spikes of processing time for predators is a result of the randomly assigned positions of Animat agents during the growth phase where predators will be required to search further for prey or partners as opposed to the closer proximity of each during the stage where both species have filled out the environment.

The results of the serial implementation on the i7 CPU shows that at around 300,000 agents, the typical Animat model has reached standard density at which individuals have closer proximity and begin to form clusters to propagate through the whole environment. On all of the architecture implementations presented, the predator agents tend to take more time than the prey to update due to the predator's capability of searching larger areas of the environment. This is exaggerated by the

predator's natural behaviour to hunt prey whereas the prey does not need to look far for grass to graze. On the GPU architectures, it can be seen that the overhead of launching CUDA kernels is costly for small population sizes under 150,000.

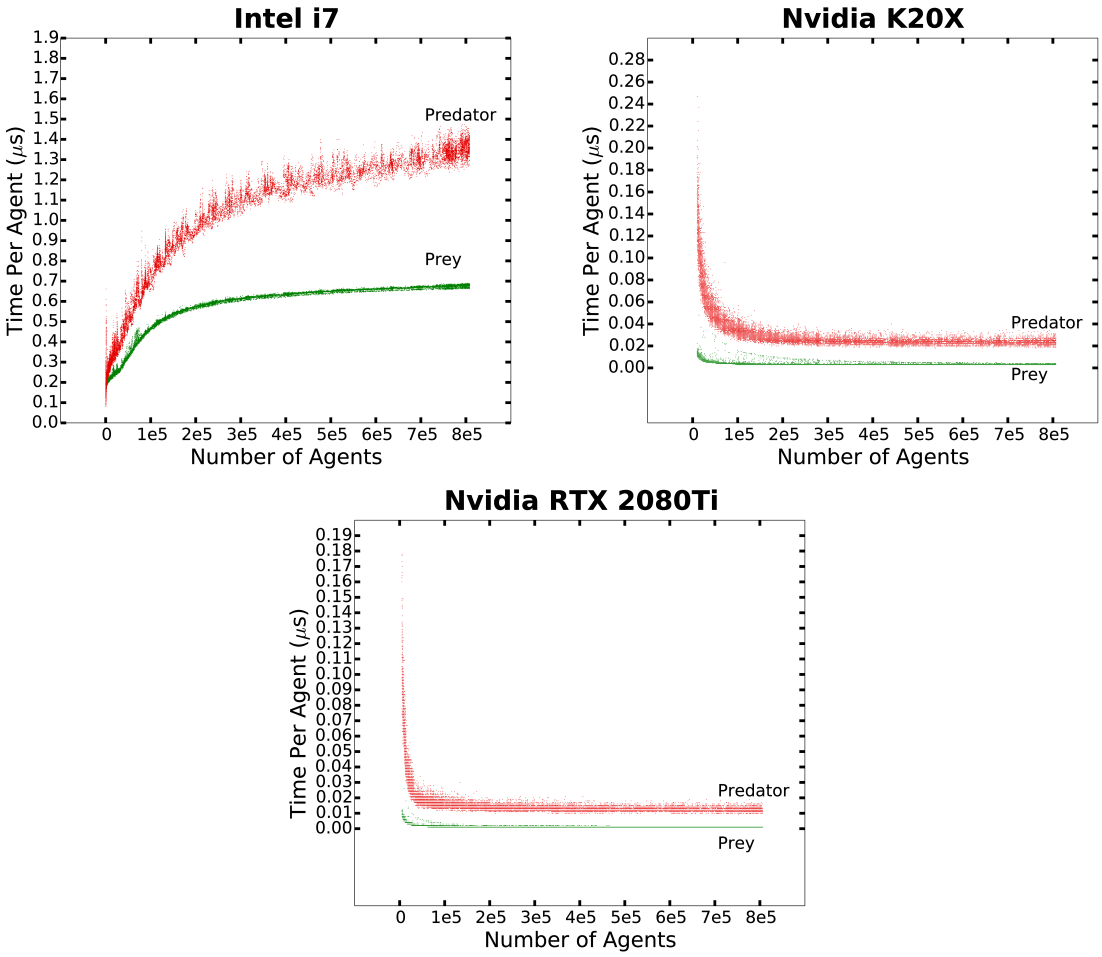

Figure 6.7: Time in microsecond per agent of the prey and predator species during the growth phase in a system size of 1500^2 for Intel i7, multi-core AMD, K20x and 2080Ti.

The average time in microseconds for the multi-core implementation is shown in Figure 6.8. When there are more available cores to process the Animat agents concurrently, less time on average is required to update the agents. However similar to the performance results shown in Section 6.1 when 32 cores are used the resource contention hinders the speed up and does not noticeably improve performance over 16 cores.

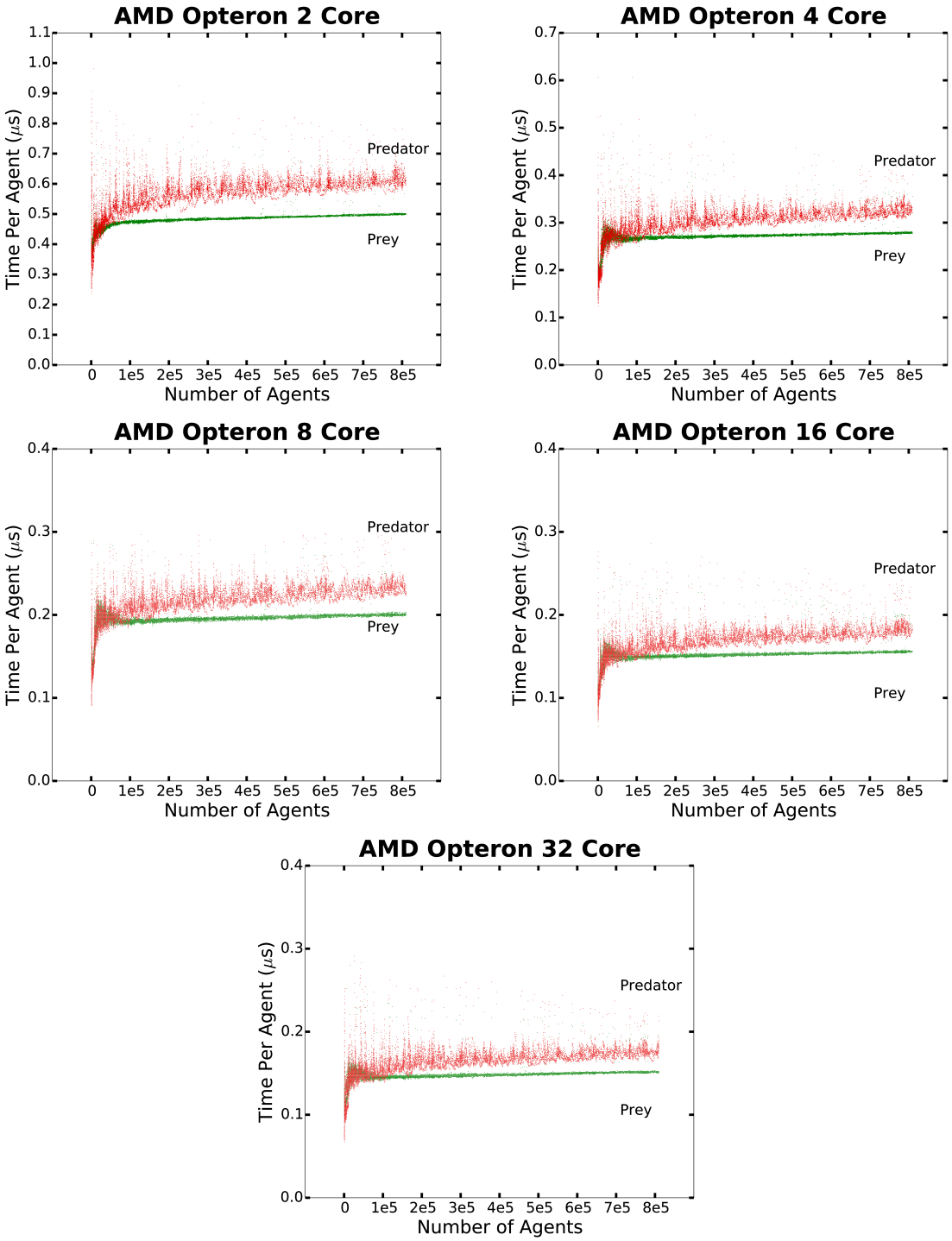

Figure 6.8: Time in microsecond per agent of the prey and predator species for multi-core AMD. Predator time increases along with the number of agents due to the initial cluster of Animat agents expanding to fill their environment as they start around the center. During this stage there are more predators that do further searches which is an expensive part of the update.

A system size of 5000^2 is used to collect the average time required to update the simulation for N number of agents. This larger system size is used to test the GPU implementations at higher occupancy. This test starts with 2000 prey and 1000 predators and allows the simulation to update to the carrying capacity of the system size which is around 10,000,000 agents with the results shown in Figure 6.9. Larger system sizes are infeasible to collect performance results using the CPU implementations as seen with the scaling of the computational costs shown in Figure 6.2.

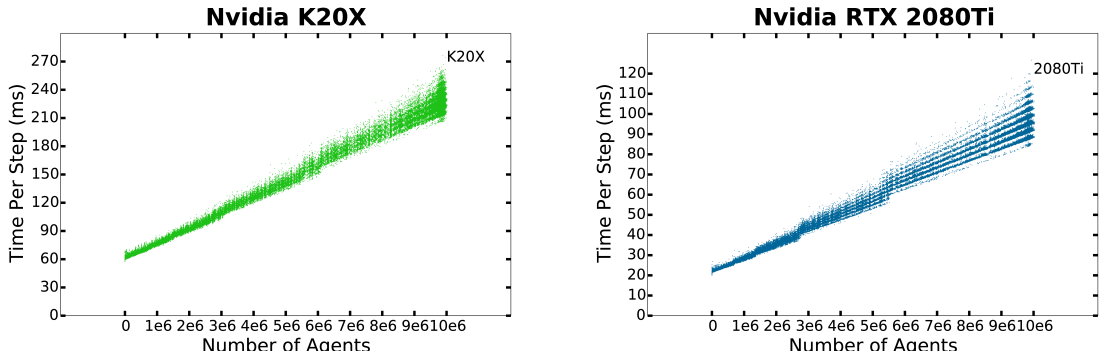

Figure 6.9: Time in milliseconds per step to update N number of agents on the K20x and 2080Ti for larger system sizes with 10,000,000 agent carrying capacity. System sizes of 5000^2 and greater allows over 10,000,000 agents to exist in a time-step.

The interesting grouping patterns or bands of the scatter plots presented in Figure 6.9 is predominantly noticeable when more agents are updated. This is caused by the increase in the number of conflict resolution kernel iterations required as the population increases. This is less noticeable in the K20X due to the overhead of other kernels that are included in the timing.

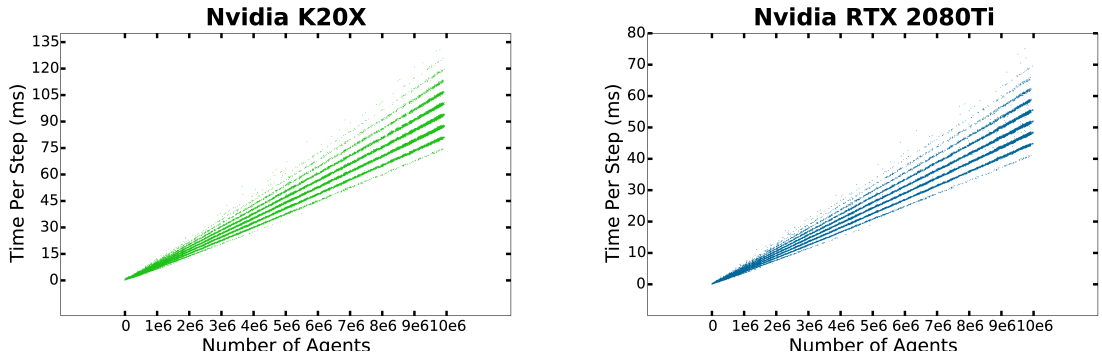

Figure 6.10: Time in milliseconds per step to resolve conflicts for N number of agents on the K20x and 2080Ti for larger system sizes with 10,000,000 agent carrying capacity.

To isolate the banding patterns more distinctly, the test is repeated with only the timing of conflict resolution kernels collected which is presented in Figure 6.10. This test shows defined patterns of time taken to execute various iterations of the conflict resolution kernel on the K20X and 2080Ti. To analyse the banding patterns, results are collected for the average number of conflict iteration calls

and the average time taken to resolve conflicts per update cycle for each system sizes. The results are presented in Figure 6.11 and Figure 6.12 which shows that the average number of conflict resolution kernel iterations remains the same on both the K20X and 2080Ti as the factor to determine the number of iterations is the proximity and the number of agents within interaction range.

The time to execute the iterations is around two times as fast on the 2080Ti over the K20X and is expected due to the performance capabilities of the newer 2080Ti GPU such as faster atomic operations, faster memory transactions and higher throughput. Fluctuations in the amount of conflict kernel iterations are due to the dynamics of the model such as the boom-bust population cycles of a predator-prey system.

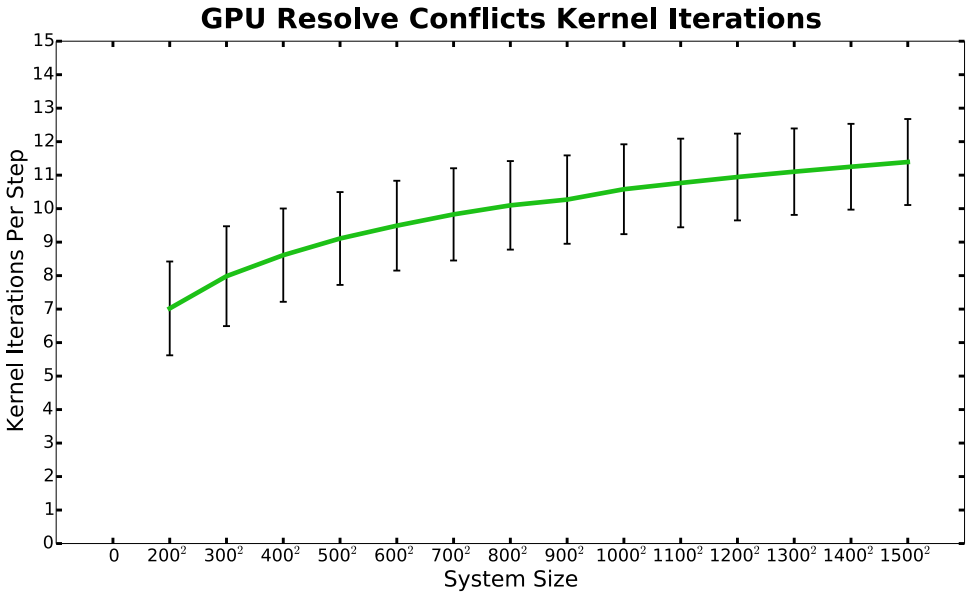

Figure 6.11: Average number of conflict kernel iterations per step for system sizes 200^2 to 1500^2 on K20x and 2080Ti.

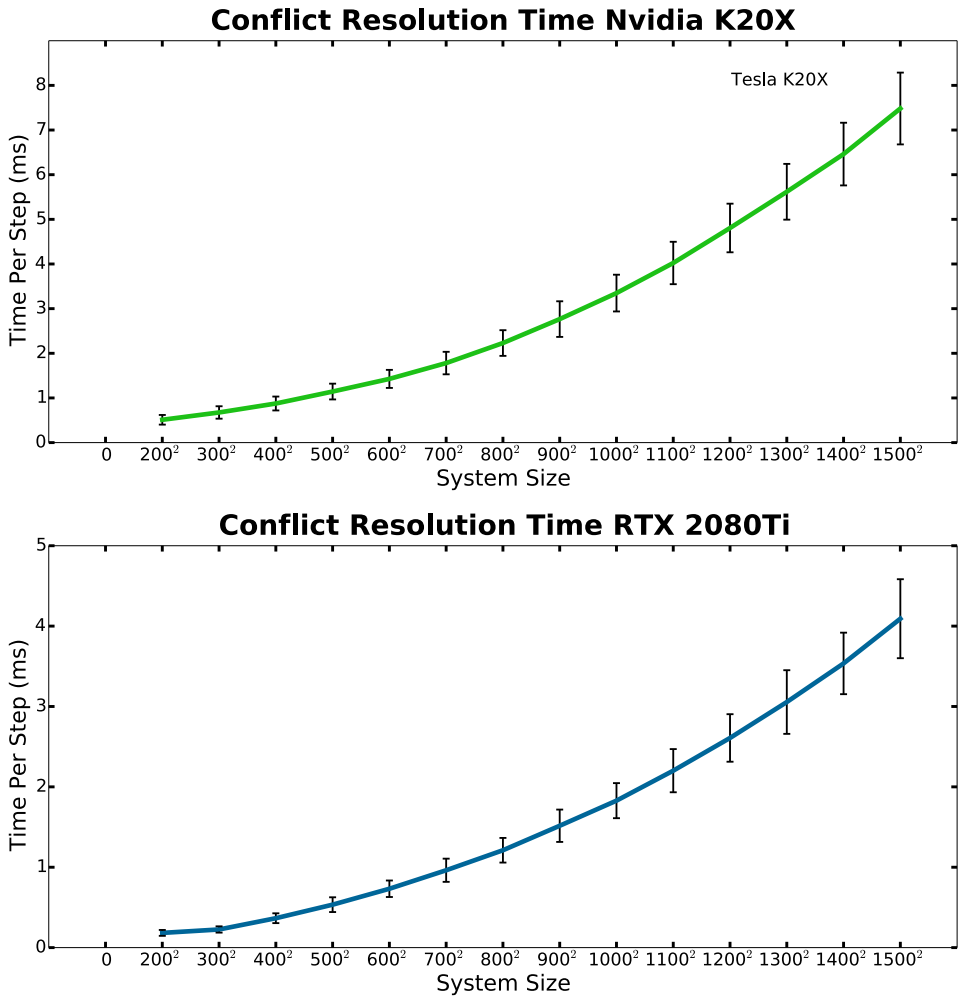

Figure 6.12: Average time to compute the conflict kernel per step for system sizes 200^2 to 1500^2 on K20x and 2080Ti.

6.3 Update Cycle

This section presents the results collected for the identified parts of an update cycle for a typical Animat model. The three parts tested are the randomising of agents for an update, the update of agents and the update of the simulation containers. Implementations for each of the architectures are discussed in Chapter 3 for the CPU, Chapter 4 for the MT-CPU and Chapter 5 for the GPU.

The stacked bars graph seen in Figure 6.13 shows the average total computational cost and the time in milliseconds to process the randomise, agent update and agent storage maintenance requirements on the CPU architectures with an 800^2 system size. The results for the multi-core implementation includes the average time required to schedule the threads for each of the processes and the initialisation of SDD tasks for a scheduled update. As expected when more cores are avail-

able the time on average to allocate memory for each task becomes lower as the threads can perform this process independently. The time required to schedule the tasks is proportionately small and allows the threads to spend the majority of time on updating agents. The use of 32 cores introduce additional resource contention and may not add more performance benefits over the 16 core results while incurring extra scheduling tasks as a result of oversubscription. This should not be confused with hyper-threading as there are 32 physical cores used in these tests and not 16 physical cores that address two virtual cores each.

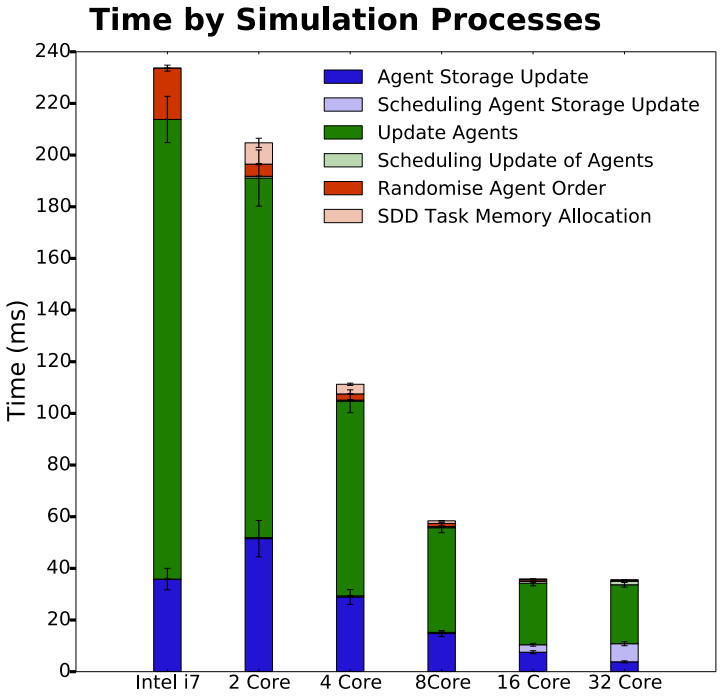

Figure 6.13: Time to complete each process per step for system size 800^2 on Intel i7 and AMD N-Cores.

A comparison of the K20X and 2080Ti is shown in Figure 6.14 where a larger 5000^2 system size is used to collect results of the GPUs during higher occupancy. The results show that using the same implementation on newer GPU hardware with higher performance capabilities can improve the performance of the simulation. This suggests the implementation developed in this thesis for GPU architectures can be used on future releases of GPU hardware without significant refactoring of code.

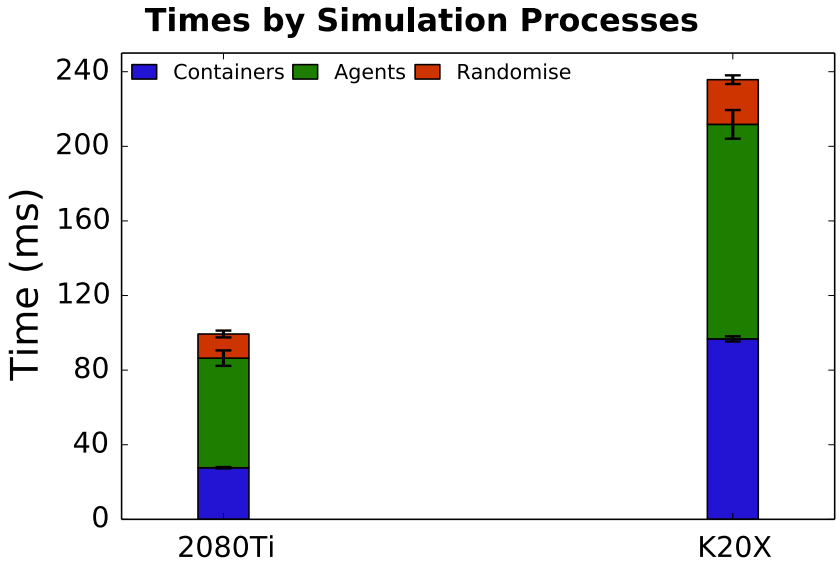

Figure 6.14: Time to complete each process per step for system size 5000^2 on RTX 2080Ti and K20X.

Randomising the order of agents for an update on the GPU implementation can be achieved with different approaches to produce a permutation of priority numbers for simulation. Two approaches used in this thesis are compared, the first performs the shuffling of priority numbers on the host during the device update of the simulation and the second uses a parallel algorithm to permute the priority numbers on the device. These algorithms are discussed in Chapter 5.

Figure 6.15 plots the time taken in milliseconds to permute the priority numbers for N number of agents up to 800,000 using the two approaches. The CPU shuffle increases at a higher rate than the dart permutation as more priority numbers are required for more agents. Although the CPU shuffle can be performed concurrently with the simulation, the priority numbers must be permuted on separate threads, streams and container locations which require additional thread coordination, synchronisation and device resources.

The scaling of the dart permutation suggests that the proportion of time required in comparison with the overall time shown in plots in Figure 6.6 are relatively small when more agents are added. The approach of dart throwing to produce thread indices to use as priority numbers is not shown as the time to compact the dartboard is around 0.05 milliseconds for 800,000 agents which is insignificant compared to the time taken to throw darts.

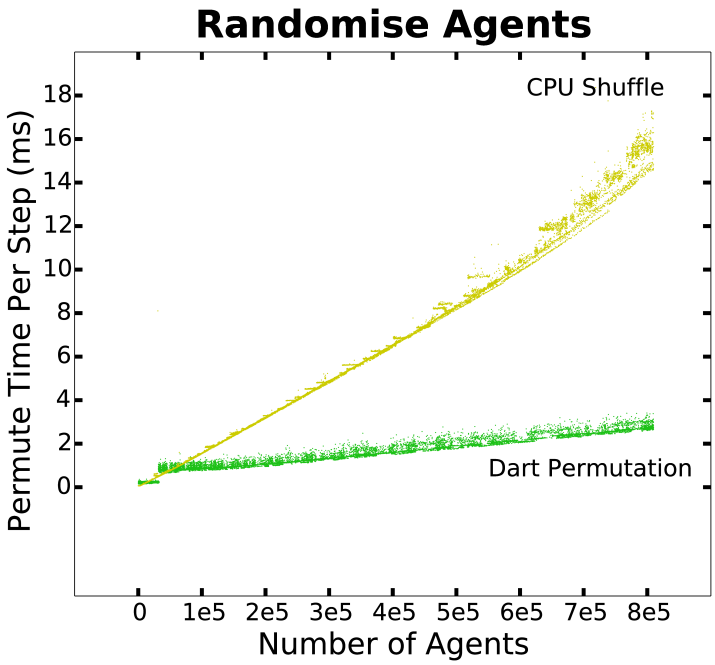

Figure 6.15: Time to randomise N number of Agents for update using the C++ threaded CPU shuffle on the i7 CPU and Dart permutation on the K20X GPU.

6.4 Replication of Original Model Behaviour

The importance of reproducibility is the benefit of aiding analysis to understand results from experiments. Reproducibility also plays a significant factor in debugging ABMs by producing the same results to correct simulation anomalies such as agents behaving incorrectly. The challenging aspect of porting a serial implementation to a parallel framework is to retain behaviours emerging from stochastic simulations. The challenges become present when applying parallelisation strategies to serial code. Pseudorandom number generators (PRNG) are used to produce stochastic predator-prey models. In an Animat model where agents are updated sequentially, re-using PRNG seeds will reproduce the same simulation results.

There are various ways to confirm a successful replication of stochastic ABMs. The standard approaches are to completely match the simulation state or use some other statistical analysis appropriate for the simulation data. An objective of porting the serial version on CPU architecture to a high performing GPU CUDA platform is to maintain model behaviour and improve performance.

Replication of the original model behaviour in this thesis uses the approach of matching all model simulation data from the CPU to the GPU implementation over an adequate amount of steps. Data that has been compared included the state of the agents, the actions performed and population numbers per-step. It is tested that the GPU implementation of the Animat model is in fact modelling the same behaviour as the original model on the CPU. The state of Animat agents, the actions

performed and the population numbers have been observed to match over 100,000 simulation steps. This number of update cycles is reasonable as in a typical Animat model simulated on a 600^2 system size; there are on average 200,000 live agents per step that could potentially interact.

The simulation states on the CPU and GPU implementation are identical when specific configurations such as the model parameters, initial conditions and PRNG are the same. The PRNG must also use the same seed to generate the same sequence of random numbers. The GPU implementation of the Animat model uses a parallel PRNG which is different to the serial PRNG used for the serial implementation of the simulation. Both versions of the PRNG are valid and is the only difference between the two implementations. If the same PRNG with the same seed is used for both implementations then all the simulation results are exactly the same.

The parameters perturbed do not introduce any biases and serve to confirm replication. A test to confirm reproducibility of the original model behaviour can be configured as follows:

The XORWOW is used in both the CPU and GPU for the random number generation requirement of the model. Both implementations are initialised using the same sequence of seeds for each of the random number generator(RNG) states. These states are assigned per agent on the CPU to match the computational model of the GPU.

The environment dimensions must match as even an extra cell in any of the X or Y dimension can propagate deviances in agent behaviours such as movement. In both implementations, an arbitrary system size of 700 by 700 is used, consisting of a typical 600 by 600 central grass area and 100 non-grass boundaries.

The update order of Animat agents per step is implemented differently on the CPU and GPU platforms. The CPU version utilises a temporary container of pointers to Animat agents in heap memory that can be shuffled and iterated to give a random order. This iteration sequence of the shuffled Animat agent pointers governs the agent behaviour and state for the next step. The GPU implementation, for performance purposes, utilises a priority number system discussed in Chapter 5 to achieve a permutation of the update sequence for processing in parallel. To completely match this process on both architectures, the CPU version is slightly modified to use an array of priority numbers that are unique and assigned to each Animat agent for processing in a serial sequence. The shuffling of priority numbers on both architectures are performed on the host and uses the C++ standard template library (STL) shuffle function which is passed the STL Mersenne Twister MT19337 random number generator with the same seed.

The searching pattern used in the replication test is a fixed sequence closest to farthest by cell distance. While a random search pattern could have been used, extra memory for RNG states is required and makes no difference to reproducing the simulation results. The crucial aspect is that the search pattern chosen must be implemented for both architectures. A simple example is to observe the processes of a fixed and random search pattern, although both consider nearest cells first they may choose different directions when two cells in each of the directions are of the same distance.

The fixed pattern would observe a particular direction first for equal-distant cells whereas the random pattern would use a random number to determine which cell to choose. This minor difference is enough to cause variations in simulation states per step. If the same predator on both architectures consumes different prey, then one prey on one implementation may have died but bred

on another. Minor differences in each update cycle will propagate the changes and result in a different simulation outcome.

The data containers for both architectures are stored differently to optimise memory transactions. On the CPU implementation, the Animat agents are stored in a grid map using agent pointers to create a singly-linked-list for direct access at each cell location. On the GPU implementation, a compressed sparse data structure is used to store Animat agents in a sorted by location order.

To guarantee the order of agents occupying cells on both implementations, a preprocessed sorting of agents by their location and then their id is performed. The CPU implementation can be modified to match the storage ordering of the GPU version by maintaining the grid of linked-lists in a sorted by ID order. Sorting in this sequence allows agents occupying the same cells to be searched in the same order across both architectures. The ordering of multiple Animat agents in the same cell are sorted by ID for replication purposes and can be arranged in any user-defined configuration. The order in which agents are added to this linked list does not change the behaviour of the model.

6.5 Individual Environment Search Patterns

The results presented in this section are collected from tests with ranged search patterns as a strategy for agents to use when deciding movement directions. The first search pattern involves agents searching all of their vision areas to find the direction with the most target agents to move towards. The second search pattern also requires an individual environment search; however, the direction is decided by probability from the density and range of occupied cells within vision. More details of the search patterns can be found in a publication [179]. The system size used for these tests ranges from 200^2 to 800^2.

The results in Figure 6.16 show the computational cost of deciding on a direction with the most agents on the CPU and GPU. At a model size of 800^2 the implementation on the K20X GPU can achieve ten times speedup over the i7 CPU. The scaling of the performance suggests that larger system sizes are still feasible for the GPU architectures to explore. Figure 6.17 shows the results of searching a direction by probability from density and range of other Animat agents on the i7 CPU and K20X. The requirements of these search patterns are similar in the aspect that they both require a lookup of all cells within vision for the agent to determine their decision.

The 2080Ti compared to the K20X provides around eight times speed up for the search patterns used in this test. The peak performance of floating point operations per second of the 2080Ti is around 3.5 times higher than the K20X. An attempt to analyse the kernel metrics of the 2080Ti and K20X is impacted by the capabilities of profilers available to profile these two compute architectures of 3.5 and 7.5. At the time of writing this thesis, compute cards of version 5.0 and lower can only be profiled using NVIDIAs NVPROF and for compute versions 5.0 and higher, the NVIDIA NSIGHT Compute profiler must be used.

The metrics for 100 replays of the predator search prey CUDA kernel which takes the most device time to compute is shown in Table 6.1. These metrics include the achieved occupancy, which is the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor in percentages and the number of memory transactions to various types of device

Table 6.1: Profiled metrics for predator search prey load and compute CUDA kernel on K20X and 2080Ti

	Achieved Occupancy %	Shared Load Transactions	Shared Store Transactions	Local Load Transactions	Local Store Transactions
K20X	49.69	97427042	432762	50588976	50813251
2080Ti	98.00	116176291	452880	34024958	34047845

	Global Load Transactions	Global Store Transactions	Device Memory Read Transactions	Device Memory Write Transactions
K20X	949447	107239	647965	5526741
2080Ti	506144	35152	552191	1236521

memory. A description of the GPU memory types is provided in Chapter 1.

The eight-times speedup of the 2080Ti over the K20X to update a cycle of the ranged search patterns is more than the peak capabilities of the 2080Ti over the K20X. This is due to factors such as the number of registers required for the local variables and the high precision type primitives such as doubles used for calculation of probabilities. There is a maximum number of 255 32-bit registers per thread, and double precision types require two registers per variable. When the kernel needs more registers than there are available per block, the CUDA application may attempt to reuse registers or spill them to other memory types. The spilling process may eventuate to off-chip device memory which is the high latency global memory that is costly to instruct repeated transactions. The spilling process requires device resources to move data through the caches and memory resulting in fewer resources available for actual computation.

The 2080Ti has a 96KB unified L1 cache and a 6.1MB L2 cache, and the K20X has a 64KB L1 and 1.5MB L2 cache. The 2080Ti has better hardware to contain spills and the amount of read and write transactions seen in Table 6.1 also suggests this. Another process within the Kernels that may require the use of local memory (global memory) is the square root function which may use the slow path to achieve correct results. The slow path is an attempt to reduce register pressure by storing some intermediate values in the local memory. The predator search prey CUDA kernel is presented in the Appendix A and each search strategy also requires a similar CUDA kernel for predators to search for other predators which can be just as costly.

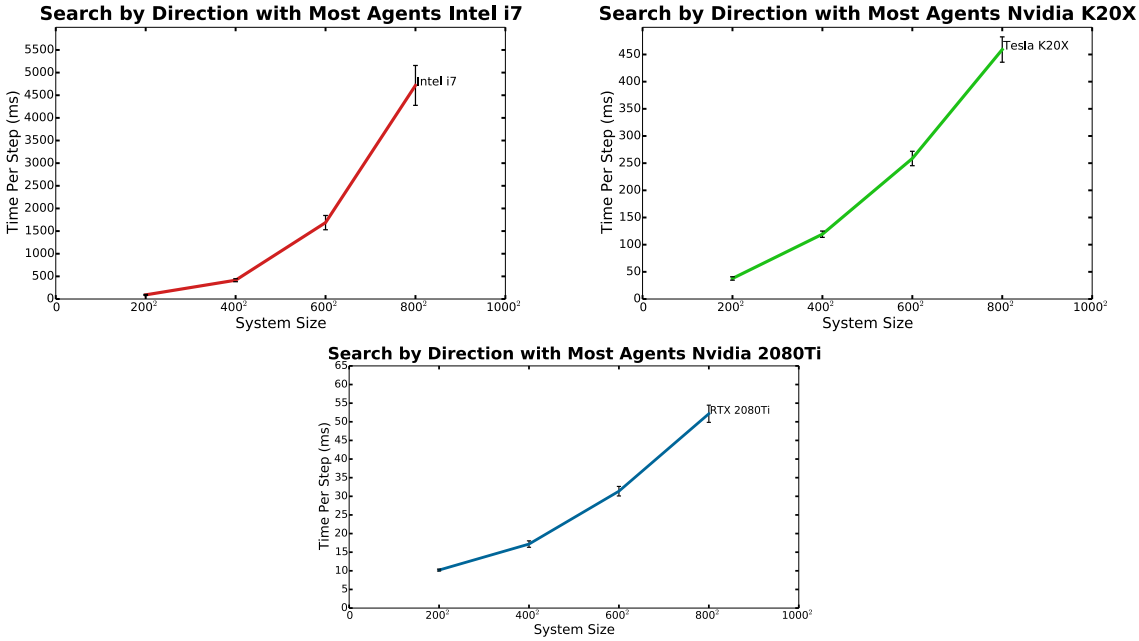

Figure 6.16: Average time per step in milliseconds for search by direction with most agents on system sizes 200^2 to 800^2.

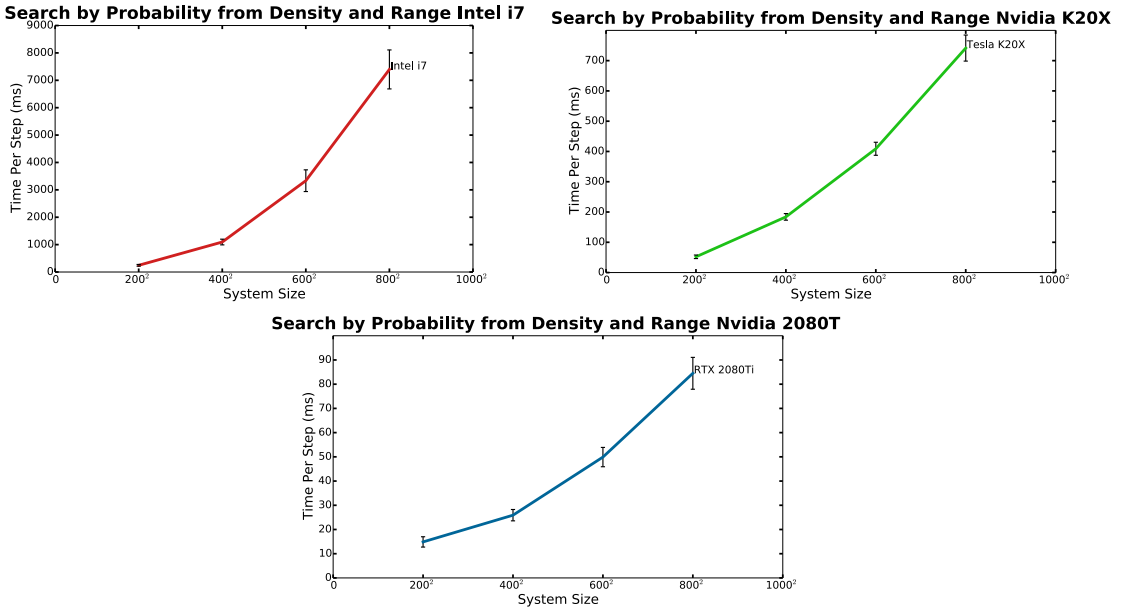

Figure 6.17: Average time per step in milliseconds for search by probability from density and range on system sizes 200^2 to 800^2.

6.6 Summary

This Chapter presents the performance results of the Animat model implemented on the CPU, Multi-core and GPUs. System sizes ranging from 200^2 to 1500^2 are tested to compare the computational costs of simulating the Animat model on these architectures. Data for larger system sizes ranging from 2000^2 to 5000^2 are presented for the GPU architectures to compare the performance at high occupancy for the devices. A break down of the standard update cycle requirements across the architectures is compared and discussed and the process to reproduce and replicate serial model behaviour is presented. The performance comparison of search patterns used by agents to decide on a movement direction is presented. This shows that the performance of computationally expensive additions to the Animat model can be improved when implemented on the GPU.

7

Agent Definition with CUDA Lambda

Animat agent definition refers to specifying the behaviours of an individual. A typical implementation of the Animat model has a fixed set of rules that is initialised by the model configuration. The rules and priority of a ruleset is discussed in Chapter 2, and these rules may belong to an individual or a species. There are no restrictions on how rules are defined; they are entirely formulated by the expert. The expert may introduce rational or irrational rules, and the definition of these rules may cause the system to become unstable or exhibit emergent phenomena. Unstable systems typically result in collapsed populations.

Experimental device Lambdas used with template CUDA kernels can assist users in defining agent behaviour by writing Lambda functions that can be provided to pre-implemented CUDA kernels. They can be used in the place of function pointers to pass device functions to other device functions.

The definition process involves the programmer choosing pre-implemented device action functions in which the requirements such as the condition or information related to the rule must be specified. This process creates a rule, and these rules can be combined and permuted in any order to create a rule set. Agent definition includes specifying model update requirements such as the health toll and the rate at which individuals age in an update cycle.

The experimental host and device Lambda features were released with CUDA 7.5 to introduce the use of Lambda functionality in CUDA applications. The experimental feature assists the process of defining individual agents, the implementation of dynamic individual rulesets and the modifications or maintenance of the simulation by removing semantics such as the switch-case conditional statements that require the user to understand CUDA mechanics when adding functionality.

The implementation of dynamic rulesets described in Chapter 3 can be achieved using function pointers to anonymous functions with C++. Previous to CUDA version 7.5, these features were not available and multiple uses of switch-case statements are required to achieve dynamic individual rule set configurations.

The agent definition on the CPU implementation utilises an array of function pointers that is iterated to call the corresponding condition and action functions of the ruleset. This is accomplished on CUDA by enabling the experimental Lambda feature to define a selector Lambda data type that can be passed to a CUDA kernel for use on the device. The selector Lambda that is provided to the

CUDA kernel is similar to an array of function pointers and can simplify the configuration of model requirements such as evolving rule-priorities of Animat agents [197].

7.1 Selector Lambda Data Structure

The selector Lambda data structure has a similar mechanism to an array of function pointers and can be implemented using the capture clause to provide a group of device functions that can be accessed by CUDA kernels through indexing. The capture clause allows the device Lambda expression to capture variables and importantly for this case, other device Lambdas.

7.1.1 Action and Condition Lambda

An Animat agent's behaviour is implemented as a set of rules that consist of actions and their conditions. Defining behaviour using Lambda expressions is accomplished using the bottom-up approach in which action and condition Lambdas are written to be captured by a selector Lambda.

Figure 7.1 visualises how the device condition Lambda can be paired with a device action Lambda to create a rule Lambda for the ruleset. The update of health and age toll is fixed for each species in a typical Animat model, however, including this update in the Agent definition process allows it to belong to an individual. A user can reuse a condition Lambda for one or more action Lambda, for example, condition A can be used for Action A, B or C if the parameters and requirements relate.

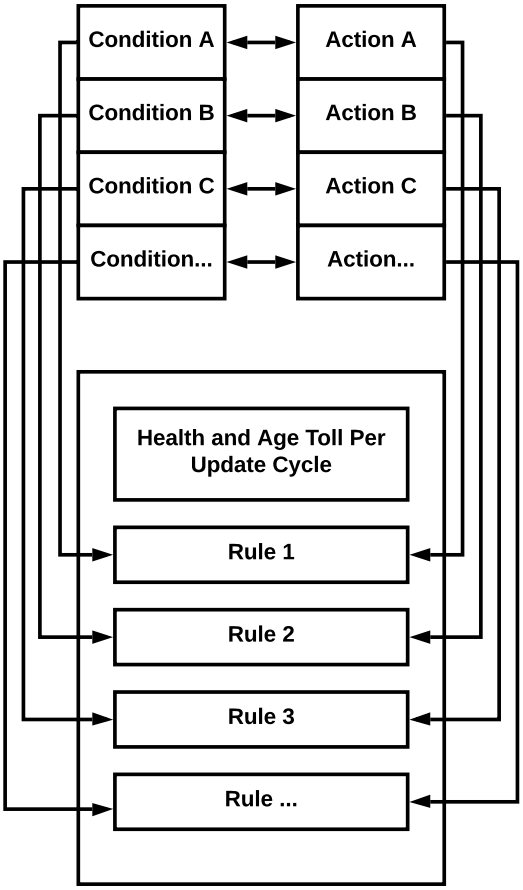

Figure 7.1: Visualisation of condition and action device functions that are captured and combined to form rules.

7.1.2 Selector Lambda

Figure 7.1 visualises the callable inline device functions in the action Lambda that are captured to create the selector Lambda. Each selector component is separated into parts that are expressed as Lambda layers. Each Lambda in layer one is developed by defining the parameters required by the encompassing inline device function. Layer two involves the capture of the defined Lambdas from layer one and is combined to create a selector Lambda. The selector layers can be used to group conditions and actions that are passed to CUDA kernels for access.

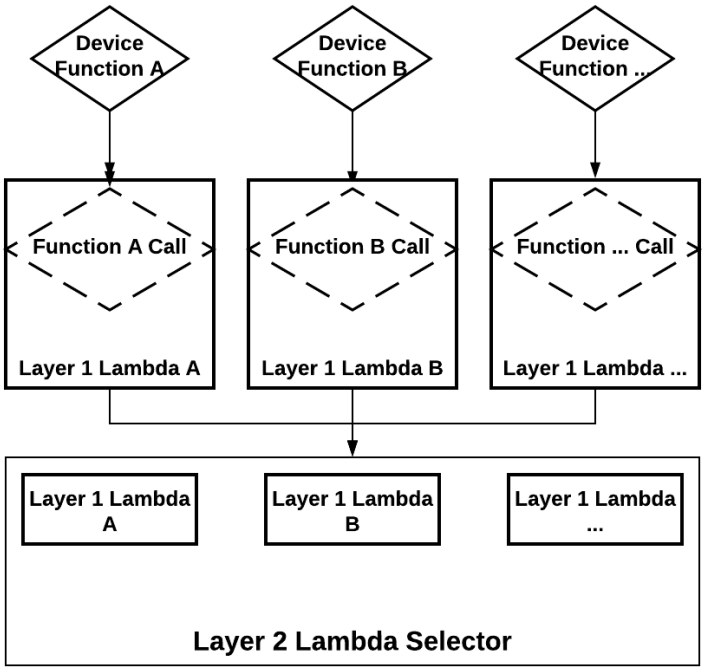

Figure 7.2: Visualisation of selector Lambda layers.

7.2 Implementation

Extended host and device lambda functions are a recent experimental CUDA feature that can be used to create expressions to capture out of scope Lambdas or device functions. Template kernels are constructed that work with device Lambda functions to allow dynamic agent definitions. The experimental Lambda feature must be compiled using NVCC with the –expt-extended-lambda flag on a platform version CUDA 7.5 or later. The type of the Lambda requires the use of the auto specifier as it is compiler generated and therefore has an unknown type. An added benefit of using a Lambda for inline device functions in Layer one is that each captured Layer one Lambda function can have a varying number of parameters and types which is not the case with an array of function pointers.

The action, condition and selector Lambdas can be written within the `main()` application function, which means the expert does not have to thoroughly understand the underlying CUDA code but rather just the provided device functions that can be read like any other C/C++ function.

Listing 7.1: Condition Lambda for Eat Rule

```
1 auto pred_decide_eat_cond = [] __device__ (int max_health, int updated_health, int
      prey_neighbour_count){
2     // If predator is hungry and there is neighbouring prey to eat
3     return (prey_neighbour_count != 0 && pre_hunger_wellfed_check(max_health, updated_health
          , 5, 0)) ? true : false;
4 };
```

Listings 7.1 is an example device Lambda function for the eating condition of the predator

species in Layer one named `pred_decide_eat_cond`. This condition Lambda checks if the predator and its current situation satisfy the condition of the rule by checking if there is prey nearby to eat and also if the predator is hungry enough by calling the `pre_hunger_wellfed_check`. The parameters of the check are set by the expert, and in this case, the parameters `5` and `0` means the predator's health value must be below 50 per cent with no restrictions for the health to above a certain level. The algorithm to check the hunger and well-fed level is described in Chapter 3. The `pred_decide_eat_cond` Lambda will be deduced to a true or false boolean to signal if the predator has satisfied the condition.

Listing 7.2: Action Lambda for Eat Rule

```
auto pred_eat_act = [] __device__ (DeviceState dPreyState, DeviceState dPredState,
    DeviceHelperState dHelperState, int tid, int move_number_vec_index){
   int position    = dPredState.d_location_ptr[tid];
   int id          = dPredState.d_id_ptr[tid];
   int move_number  = dPredState.d_move_number_ptr[move_number_vec_index][tid];
   int eating_index = dPredState.d_eaten_by_ptr[tid];
   int eating_id   = INT_MAX;

   // Check if first time trying to eat if not get the eaten_by number of last chosen prey
   if(eating_index != INT_MAX){
      eating_id = dPreyState.d_eaten_by_ptr[eating_index];
   }
   // If predator caused a conflict or hasn't found food
   if(eating_id != move_number){
      dPredState.d_eaten_by_ptr[tid] = INT_MAX;
      eating_index = INT_MAX;
      // Predator finds prey to eat which may have raised a conflict
      if( eat_prey_conflicts_b(position, id, move_number, eating_index, dPreyState,
          dPreyState.d_start, move_number_vec_index)){
         *dHelperState.d_eat_conflict_flag = true;
      } // Temporarily write index to consumable prey
      if(eating_index != INT_MAX){
         dPredState.d_action_ptr[tid] = 'E';
         dPredState.d_eaten_by_ptr[tid] = eating_index;
      }
      else{ // Failed to locate edible prey
         dPredState.d_action_ptr[tid] = '?';
      }
   }
   return (dPredState.d_action_ptr[tid] == 'E') ? true : false;
};
```

Listings 7.2 is an example implementation of the predator eat rule. Although this is a very low-level implementation, it allows the expert to specify what happens after a predator has found food. The predator may decide to consume the prey itself or exhibit an altruistic behaviour by sharing the resource with neighbouring predators [66]. The expert may not require the low-level abstraction and can write the Lambda to call another function that restricts how the food resources are assigned.

Listings 7.3 is an example selector Lambda for the conditions of the predator species ruleset from the typical Animat model. This condition selector Lambda can be pair with the action selector Lambda implementation as shown in Listings 7.4. The pairing of the selector Lambda represent the predators rule set and selecting a rule from the selector Lambda only requires an integer to be provided to the `condition_index` and `action_index` parameter.

Listing 7.3: Predator Condition Selector Lambda

```
auto pred_conditions = [pred_seek_mate_cond, pred_seek_prey_cond, pred_decide_eat_cond,
    pred_breed_cond, move_randomly_cond] __device__ (DeviceState dPreyState, DeviceState
    dPredState, DeviceHelperState dHelperState, int tid, int condition_index){

   // Load require information
   int updated_health       = dPredState.d_health_ptr[tid] - 10;
   int location_k           = dPredState.d_location_ptr[tid];
   int pred_neighbour_count = dPredState.d_neighbour_count[location_k] - 1;
   int prey_neighbour_count = dPreyState.d_neighbour_count[location_k];
   int max_health = cSpeciesData[pred_index].max_health;

   // Provide index to captured condition Lambdas
   if(condition == 0){
      // If the decide to eat condition failed
      if(!pred_decide_eat_cond(max_health, updated_health, prey_neighbour_count)){
         dPredState.d_action_ptr[tid] = '?';
         return false;
      }else{ // If condition is satisfied and there are neighbouring prey to eat
         return true;
      }
   }else if(condition == 1){ // Breed condition
      return pred_breed_cond(max_health, updated_health, pred_neighbour_count);
   }else if(...){ // Other conditions
      return ...();
   }
   else{ // All decisions invalid
      return false;
   }
};
```

Listing 7.4: Predator Action Selector Lambda

```
auto pred_actions = [pred_seek_mate_act, pred_seek_prey_act, pred_eat_act, breed_act,
    randomly_move_act] __device__ (DeviceState dPreyState, DeviceState dPredState,
    DeviceHelperState dHelperState, int tid, int action_index, int move_number_vec_index){
    // Provide index to captured Action Lambdas
    if(action_index == 0){
      return pred_eat_act(dPreyState, dPredState, dHelperState, tid, move_number_vec_index)
          ;
    }else if(action_index == 1){
      return breed_act(dPredState, dHelperState, tid, pred_index);
    }else if(action_index == ...){
      return ...();
    }else{
      return false;
    }
};
```

The condition and action selector pair is visualised in Figure 7.3 in which pairs of the condition and corresponding action can be written in any order. The expert can use an array of integers to specify a rule set for individuals or the species. This means switching back and forth between a dynamic and fixed rule set is a simple process of defining an array of indices. The array can be defined based on the permutation of the rule set, and the order of the indices specifies the priority of each rule. An example integer array with four rules: $3, 0, 1, 2$ would process the rules in the order: Move Towards, Breed, Eat then Move Away seen in Figure 7.3.

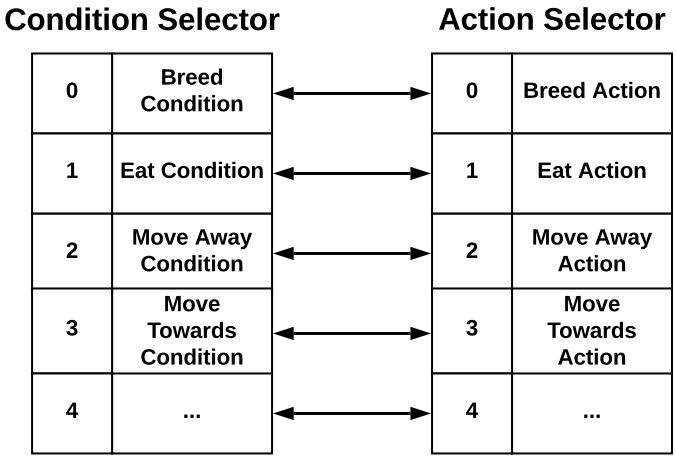

Figure 7.3: Visualisation of condition and action selector pairs.

7.2.1 Decide Behaviour Template CUDA Kernels

The use of Lambdas in CUDA requires template CUDA kernels to pass the generic Lambda selectors for use in the `__global__` kernel body. Listing 7.5 is an example Prey decide action template CUDA kernel. The selector Lambdas are specified as template parameters named Check, Condition and Action. The Check selector Lambda captures the update cycle check Lambdas which determines if an Animat agent can decide on a rule.

The check Lambdas include update cycle requirements implemented as device functions such as health and age toll update or a status check. Representing them as Lambdas allows the definition of these cycle requirements to also belong to an individual.

These template CUDA Kernels are hidden to the expert defining the agents. The kernel coordinates the control of threads of a thread block and requires the expert to provide the selector Lambdas and an index array in which to iterate and select Lambdas to process. The use of the selector Lambdas significantly reduces the amount of code required to implement and maintain this kernel and also allows the conflicting and non-conflicting CUDA kernels to be sensibly combined. For example, if the prey species has a rule that can cause conflicts, the expert can simply provide the index to this rule for the `start_rule_index` and `max_rules` parameter. This effectively processes the conflicting rule only and can be called iteratively until all conflicts have been resolved.

Listing 7.5: Template Prey Decide Action Kernel

```
template <typename Check, typename Condition, typename Action>
__global__ void prey_decide_act(Check checks, Condition Conds, Action Acts, DeviceState
    dPreyState, DeviceState dPredState, DeviceHelperState dHelperState, int max_rules, int
    start_rule_idx, int *d_prey_rule_indices, int num_checks){

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    // More threads than agents
    if(tid >= dPreyState.live_agents){ return; }

    // Number of checks before an Agent can decide on an action
    for(int i = 0; i < num_checks; i++){
        // If failed any checks then return this agent has died
        if(!checks(dPreyState, i, prey_index, tid)){
            return;
        }
    }

    // Now go through the action and its condition pair and decide on an action
    for(int i = start_rule_idx; i < max_rules; i++){
        int rule_index = d_prey_rule_indices[i];
        if(Conds(dPreyState, dPredState, tid, rule_index)){ // If condition is satisfied
            if(Acts(dPreyState, dPredState, dHelperState, tid, rule_index)){
                return; // If decision is valid
            }
        }
    }
}
```

7.3 Summary

This chapter presents an internal agent definition system that allows experts to configure agents without an understanding of the underlying parallel language. This agent definition system is developed with the experimental device Lambda features that work with CUDA template kernels. The implementation details are discussed which include the system of captured device Lambda layers to create a selector Lambda which can be used to represent dynamic individual and species behaviour.

8 Conclusions

The Animat model has been implemented on parallel computing architectures to improve computational performance whilst maintaining the original model behaviour. The parameters of the macroscopic model can be used in an abstract way to define the behaviour of individuals that belong to the predator or prey species of a microscopic system. Past and present research using the Animat model to simulate interesting systems that exhibited complex emergent phenomena is discussed in Chapter 1. A complete description and discussion of the Animat model is provided in Chapter 2.

The model consists of key components: the Agents, the behaviour and interaction rules and the environment in which they exist. The simulation of the model is progressed through an update cycle which must be analysed to identify memory access patterns to determine how the Animat components can be efficiently represented and stored. Chapter 3 describes and discusses the storage and functional requirements to simulate the model in cycles on the C++ object oriented programming language.

One purpose of the predator-prey Animat model is to investigate and understand how individuals of each species, their behaviours and interactions impact the system at the macroscopic level. These systems are often limited in size and complexity due to the computational cost of simulating these models. As manufacturers are nearing physical constraints of single-core architecture design, a shift towards multi-core production to improve performance has been seen in the last 10 years.

A strategy is required to extend the implementation of the Animat model to fully utilise the multi-core architectures and exploit the parallelism. Developing a parallelisation strategy requires an analysis of the dependencies in the model to design an update cycle for the simulation to be executed on a multi-core architecture. Chapter 4 discusses relevant parallel programming APIs and frameworks to introduce parallelism to the Animat model using a spatial domain decomposition approach.

A major component of this thesis is to investigate the use of GPU architectures to accelerate the Animat model simulation. GPUs are a common component of many devices such as mobile phones, personal computers or supercomputer clusters. They offer a great deal of computational performance and outperform consumer-level multi-core architectures for suitable parallel tasks.

One challenging factor of simulating agent-based models using GPUs is to retain the original model behaviour whilst exploiting the performance capabilities of the architecture. This has been

accomplished by developing a new update strategy suitable for GPU architectures. This strategy relies on a parallel conflict resolution method to retain original model behaviour when updating many Animat agents simultaneously. A compressed sparse data structure is built to support the new update strategy by providing storage for the model in a manner to allow for efficient memory transactions. The development of the GPU implementation and the supporting data structures are presented in Chapter 5.

The performance of the Animat model implementations on the original single-core CPU, Multi-CPU and GPU using a typical configuration across a range of system sizes are compared in Chapter 6. This chapter includes a discussion on validation of original model behaviour.

Changing the behaviour of an Animat agent may require a variety of changes throughout the program in which an understanding of the parallel update method to a certain level is needed. An implementation of the Animat agents using Lambda and template CUDA kernels allow changes to agent behaviour without significant changes to other regions of code. Using the features in this manner does not incur performance sacrifices that are significant. The implementation of Animat agent definition using experiment device Lambdas is discussed in Chapter 7.

8.1 Contributions

This thesis has shown that parallel architectures such as the GPU can be used to compute the Animat model with improved performance, and without sacrificing original behaviour.

The experimental extended host and device Lambda feature can be used to construct template CUDA kernels to improve the ease of use without loss of performance. Using the experimental CUDA features in this manner can be applicable to other models using the agent-based approach.

An update strategy relies on a method to effectively resolve conflicts caused by agent interactions for the Animat model that is not an inherently parallel problem. A solution has been presented that uses a priority number system to assign ranks to Animat agents which are used to determine a sequence of interactions. This sequence is deterministically solved by iteratively executing a CUDA kernel to resolve conflicts raised by a device function using atomic operations to compare priority numbers. The priority number system can be configured to match the update order of the original model to reproduce the behaviour on parallel architectures.

To support this system a compressed sparse data structure is developed to address the execution model of the GPU architecture and provide an efficient storage that satisfies the model requirements.

These important contributions can assist researchers using the Animat model to access high-performance GPU parallel architectures to effectively explore different simulation configurations and simulate at a larger than previously possible system scale. The method developed is highly parallel and is likely to scale well to increasingly parallel future architectures.

8.2 Discussions

Computer chip manufacturers have been able to increase the number of transistors on a single silicon chip every year since the invention of circuits around 60 years ago. The expectation of doubling the number of transistors to increase the processing power of a CPU with Moore's Law [119] has not been met over the last decade. This is due to the physical limitations of a power wall as shown with Dennard scaling which essentially states that as transistors get smaller, their power density stays constant, so that the power consumption stays in proportion with area. In 2005 two leading designers and manufacturer of microprocessors, Intel and AMD have shifted production focus to develop multi-core chips. Initial chips were constructed with dual-cores and common desktops include 6-8 or more cores with high-end chip-sets providing 32 cores.

Modern software that requires the computational performance of parallelism has driven developers into parallel programming. Parallel computing is increasingly popular for many computationally expensive tasks, for example, GPUs have featured in many commercial endeavours and scientific research such as cryptocurrency mining, modelling complex systems or investigating the deep learning approach of artificial intelligence to solve problems. However, parallel programming has always been challenging; it requires the consideration of issues such as race conditions, mutual exclusions and synchronisation when making use of threads that can range from multiple to thousands. The GPU architecture has a restricted programming model and writing efficient applications to conform is a complicated task.

Investigating complex adaptive systems with agent-based models in the field of Artificial life has been limited by the sizes and complexity of systems available for investigation. GPUs are devices that can provide powerful computation to simulate large scale models with relatively low costs. GPUs are commonly featured in high-performance computing, and as examples, the top two fastest super-computers in the world at present and around 30 per cent of the TOP100 use GPUs accelerators to achieve incredible amounts of floating point operations per second [141].

Reproducibility is an important aspect of evaluating scientific research [198, 199] and can be an issue when parallelising ABMs. A sequential version of an ABM can be reproduced by re-use of the same pseudorandom number generator as processes are scheduled deterministically. However, when reproducing ABMs on parallel architectures, careful considerations are required to replicate the system behaviour of a serial implementation. For example, an attempt to parallelise a serial model has been unsuccessful, the discussion on the processes are provided by Hazel Parry and Mike Bithel [200]. Nuno Fachada claimed that two conditions must be verified to make an ABM such as the predator-prey system used in that work, reproducible [201]. The first condition mentions that each thread must have its own PRNG sequence and the second condition is that the final state of shared memory must be deterministic and should not depend on thread scheduling.

The author also stated that the second condition is complicated to guarantee since thread scheduling is not deterministic. The author mentions that a two-step solution to register intents then deterministically solve conflicting intents to satisfy the second condition "can have a very detrimental impact on simulation performance, possibly defeating the purpose of parallelisation". The author proposes that by dropping the reproducibility requirement, conflicting thread actions disappear and

performance is not sacrificed. In the same work, simulation data of various implementations have been compared using statistical focal measurements.

The methods described in this thesis have shown that it is possible to replicate individual behaviour from serial models to reproduce system behaviour and improve the performance of the simulation on high-performance parallel architectures such as the GPU device. The method uses a conflict resolution CUDA kernel to provide deterministic ordering that can be computed in parallel. The performance results and a means to verify the reproducibility of serial model behaviour is discussed in Chapter 6.

Although GPUs have accelerated the performance of agent-based models, there has been fewer attempts to implement systems such as the Animat model which consists of individuals with behaviours that may cause conflicts during interactions. An example is the behaviour of predation in the predator-prey Animat model; it is naturally implied on a sequential model as the predator that is prioritised to act would consume the prey first. However, when addressing many threads that are typically assigned to 100,000 or more predators a system to resolve the issues is required. The concept of conflicts in the Animat model may apply to other complex agent-based models when developed for parallelism. Conflicts may occur when any type of multiple individuals require exclusive access to the same resources simultaneously. The resolution process must also reproduce the ordering from the original model to replicate the behaviour. A solution to this problem has been developed and is discussed in Chapter 5.

The introduction of search patterns [179] is an experiment on how model behaviour affects simulation performance. This addition can be combined with evasion choices in a predator-prey simulation [116] to form the kind of experiment a researcher may wish to conduct.

The foreseeable future of computer chip design is likely to continue the use of multi- and many-cores. With this in mind, the methods presented are likely to transfer to new parallel architectures and gain performance from hardware improvements without major refactoring. As new features may become available on new chipsets, modification of certain functions may be required to exploit the new features.

The methods presented in this thesis could be applied to other agent-based models as the issues they address are common problems in many systems. Individuals or agents in many models have an internal state and behaviours that result in complicated interactions. Simulating systems that are large enough to produce the desired emergent phenomena may require the use of parallel devices to provide results in a reasonable time frame.

8.3 Future Work

Definition of agent behaviour in the work of this thesis can be achieved by the use of writing device Lambda functions that make use of predefined Agent actions. This simplifies the agent behaviour definition process or the user with sufficient programming ability and knowledge to achieve this without refactoring CUDA code. The next logical goal would be to create a domain specific language (DSL) for the Animat model to accommodate researchers in other scientific fields such as ecology to design specific models without requiring programming knowledge. This would allow

the user to focus on the model without being exposed to technical implementation details they are not required to understand. The extension, in essence, would improve access to the Animat ABM to other disciplines other than computer science.

The challenge of designing an adequate DSL for the Animat model requires thorough considerations to the design to improve productivity and technical implementation details to retain performance and reproducible requirements.

In order to improve productivity, it would be beneficial to develop a DSL with the domain experts requirement in mind. The level of success of a DSL may be measured by the ease with which the domain expert can define model requirements. The DSL must easily allow the user to specify or define components of the Animat model such as the individual's attributes, behaviour, interaction and the environment. For example, the domain expert may simply define an Animat agent with high level abstraction, such as a sentence: *Agent type* is a *predator* that can exists for a 100 *cycle* and *hunts* when hungry or else *breed*. This sentence does not describe the implementation details or full information of hungry and breed etc but provides the domain keywords that can be interpreted to automatically configure the simulation.

The design would closely relate with implementation details as this may provide the specifications required to decide on the type of DSL; two possible types are internal and external. Internal DSLs will mean the language to define the Animat model will be built on top of an existing programming language. An external DSL will mean the language to define the Animat model will be separate from the development programming language and would require language parsing, code generation, etc.

Currently, the maximum feasible simulation size for the Animat model to work with is determined by the computational device. There has been no research to identify how large is large enough for the system size of the Animat model to determine if the macroscopic predator-prey systems model the same dynamics. The next viable option is to introduce the parallelism of multiple GPUs (M-GPU). An M-GPU system consists of multiple connected GPU devices that can work together to compute a task. This approach is easily accessible as motherboards in many desktops contain 2 or more Peripheral Component Interconnect Express (PCI-E) slots and each of these slots can host a GPU device. The M-GPU approach is further supported by the advancements in hardware such as the NVIDIA NVSWITCH and NVLINK to provide high-speed connectivity between the devices [202, 203]. The IBM summit, which at the time of writing, is the fastest supercomputer in the world [141] using NVIDIA's high-speed NVLINK to connect 6 Tesla V100 GPUs of a node. This supercomputer has 27,648 Tesla V100 GPUs spread over 4608 nodes connected via the dual-rail EDR InfiniBand network.

Although the communication speed between the GPUs has improved, the communication may be a bottleneck that limits performance. Implementing the Animat model with M-GPUs would have to be carefully designed to limit communication and overlap with computation where possible. This is a complicated task as the simulation must be split between devices which may require a lot of communication for synchronisation. Other complications include limitations to operations that can only be performed on a single device. The use of M-GPUs could help to extend the Animat research by providing access to larger and faster simulations of the model.

A
Appendix

Listing A.1 is the CUDA kernel and device function for a predator to search a cell within the vision that has the most prey. This CUDA kernel pre-processes and shares data for each occupied cell by reading information of all prey within vision from device global memory into shared memory. Any occupying predator agent within the cell can read from shared memory to find the direction with the most prey and save that information using a `bit` masking device function.

Listing A.1: Predator search for cell with most prey within vision CUDA kernel

```
__global__ void  set_pred_search_prey_most_load_compute_bm( int *prey_start, int *
    pred_start, int *pred_towards_prey_k, int *prey_cell_count ,SpeciesData *
    agentSpeciesData, int s_border_x, int s_border_y, curandState *rng_state, unsigned int
    *pred_towards_prey_bm){

   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   int tid = y*d_grid_width + x;

   extern __shared__ int s_target_count[];

   if(x >= d_grid_width || y >= d_grid_height) {
      return;
   }

   int bx   = threadIdx.x;
   int by   = threadIdx.y;
   int pred_s = pred_start[tid];

   int vision_radius = agentSpeciesData->vision_radius;
   // Returns number of threads that have an agent.
   int count = __syncthreads_count((pred_s >= 0));
   // If no thread in block has an agent, return
   if(count == 0) {
      return;
   }
   // Thread will load into shared memory for other cells even if cell is not occupied.
   int move_direction_most_agents[9];

   for(int i = 0; i < 9; i++){
      move_direction_most_agents[i] = -1;
   }
   // Can use other sizes if needed.
   const int y_start = blockIdx.y - 4;
   const int y_end   = blockIdx.y + 4;
```

```
33    const int x_start = blockIdx.x - 2;
34    const int x_end   = blockIdx.x + 2;
35
36    for(int bid_y = y_start; bid_y <= y_end; bid_y++){
37       // If out of bounds continue to next block index
38       for(int bid_x = x_start; bid_x <= x_end; bid_x++){
39          // If out of bounds continue to next block index
40          if(bid_x < 0 || bid_x >= gridDim.x){
41             continue;
42          }
43          // Load into shared memory, shared memory is mapped to the same size as block
                dimensions
44
45          // Calculate block offset into global memory
46          int bidy_offset = (bid_y) * blockDim.y;
47          int bidx_offset = (bid_x) * blockDim.x;
48          // Calculate index to global array to load into shared memory
49          int g_y = bidy_offset + by;
50          int g_x = bidx_offset + bx;
51          // Load prey counts into shared memory
52          s_target_count[by*blockDim.x + bx] = prey_cell_count[g_y*d_grid_width + g_x];
53          // Wait till all threads complete
54          __syncthreads();
55          // If an agent is occupying this cell and is searching
56          if(pred_s >= 0){
57             for(int s_y = 0, yy = bidy_offset; s_y < blockDim.y; s_y++, yy++){
58                for(int s_x = 0, xx = bidx_offset; s_x < blockDim.x; s_x++, xx++){
59                   // Offset from current location
60                   int dy = yy - y;
61                   int dx = xx - x;
62                   // Check if offset coordinate is in vision
63                   if( ((dy*dy) + (dx*dx)) <= (vision_radius*vision_radius) ){
64                      // Do not process neighbouring cells
65                      if( (dy >= -1 && dy <= 1) && (dx >= -1 && dx <= 1)){
66                         continue;
67                      }
68                      // Get current target count from shared memory
69                      int curr_prey_count = s_target_count[s_y*blockDim.x + s_x];
70                      // Continue if no agents here
71                      if(curr_prey_count <= 0){
72                         continue;
73                      }
74                      // check which direction this cell count belongs too by comparing
75                      int m_d_m_a_index = -1;
76                      m_d_m_a_index = ((dx < 0)  && (dy < 0))  ? 5: m_d_m_a_index;
77                      m_d_m_a_index = ((dx == 0) && (dy < 0))  ? 2: m_d_m_a_index;
78                      ... // Other directions
79                      // Check if current max for a particular direction needs to be updated
80                      move_direction_most_agents[m_d_m_a_index] = (curr_prey_count >
                            move_direction_most_agents[m_d_m_a_index]) ? curr_prey_count :
                            move_direction_most_agents[m_d_m_a_index];
81                   }
82                }
83             }
84          }
85          // Wait till all threads have finished using the shared memory
86          __syncthreads();
87       }
88    }
89    /*
90       - at this point move_direction_most_agents[] should contain max target counts
91          for each 8 directions
92       - compare for max or resolve for tied maximums
93       - Mask directions as bits
```

```
94     */
95 }
96
97 // Set bits for valid move directions (ties)
98 __device__ void set_agent_direction_bits(int *move_direction_most_agents, unsigned int &
       most_agents_direction_bits, int tid, curandState *rng_state){
99     // 8 indexes for worst case
100    int curr_highest_idx[8];
101    // Keep track number of ties
102    int n_ties = 0;
103    // Set first index;
104    curr_highest_idx[n_ties] = 1;
105    // Check if there is atleast 1 valid direction
106    bool valid_directions = (move_direction_most_agents[1] > 0) ? true : false;
107    // Compare 8 directions
108    for(int i = 2; i < 9; i++){
109       int curr_move_direction_agents = move_direction_most_agents[i];
110       // If there are ties with atleast 1 agent then increment and add
111       if(curr_move_direction_agents == move_direction_most_agents[curr_highest_idx[n_ties]]
              && (curr_move_direction_agents > 0)){
112          ... // Increment and Add
113       }
114       else if(curr_move_direction_agents > move_direction_most_agents[curr_highest_idx[
              n_ties]] && (curr_move_direction_agents >= 0)){
115          ... // Reset if there is a new max
116       }
117    }
118    // There is a valid direction so there is atleast a bit to set to 1 else leave as 0 to
          signify no directions
119    if(valid_directions){
120       most_agents_direction_bits = 0;
121       for(int i = 0; i < n_ties+1; i++){
122          SET_BIT(most_agents_direction_bits, curr_highest_idx[i]-1);
123       }
124    }
125 }
```

Listing A.2 is the CUDA kernel and device function for a predator search pattern that randomly decides on a direction to move towards based on the distribution of probabilities for each cell within vision range. The probability distribution for cells can be determined based on the number of occupying Animat agents relative to the distance to the cell.

This search CUDA Kernel pre-processes and shares data for each occupied cell by reading other agent information within vision from device global memory in a coalesced pattern to store in device shared memory. The vision information in device shared memory can be shared by any Animat agents occupying the cell to use a pseudo-random number and randomly choose a direction based on the distribution of probabilities.

Listing A.2: Predator search for prey by probability form density and range CUDA kernel using the load and compute approach.

```
__global__ void  set_pred_search_prey_cell_prob_lc( int *prey_start, int *pred_start, int *
    pred_towards_prey_k, int *prey_cell_count ,SpeciesData *agentSpeciesData, curandState *
    rng_state){

   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   int tid = y*d_grid_width + x;
   // Use same size as blocks
   extern __shared__ int s_target_count[];

   if(x >= d_grid_width || y >= d_grid_height) {
      return;
   }
   // Block thread id y and x
   int bx   = threadIdx.x;
   int by   = threadIdx.y;
   // Number of predators
   int pred_s = pred_start[tid];
   // Returns number of threads that have an agent.
   int count = __syncthreads_count( (pred_s >= 0));
   // If no thread in block has an agent, return Note* do not return if shared memory
   if(count == 0) {
      return;
   }
   // Contains the accumulative n_agents / distance for each move direction
   double direction_group_most_agents[9];
   // Default
   for(int i = 0; i < 9; i++){
      direction_group_most_agents[i] = 0.0;
   }
   double sum_ni_di = 0.0;
   const int y_start = blockIdx.y - 4;
   const int y_end   = blockIdx.y + 4;
   const int x_start = blockIdx.x - 2;
   const int x_end   = blockIdx.x + 2;

   // Vision radius used to search
   int vision_radius = agentSpeciesData->vision_radius;
   for(int bid_y = y_start; bid_y <= y_end; bid_y++){
      // If out of bounds continue to next block index
      if(bid_y < 0 || bid_y >= gridDim.y){
         continue;
      }
      for(int bid_x = x_start; bid_x <= x_end; bid_x++){
         // If out of bounds continue to next block index
         if(bid_x < 0 || bid_x >= gridDim.x){
            continue;
         }
         // Load into shared memory, shared memory is mapped to the same size as block
             dimensions
         // Calculate block offset into global memory
         int bidy_offset = (bid_y) * blockDim.y;
         int bidx_offset = (bid_x) * blockDim.x;
         // Calculate index to global array to load into shared memory
         int g_y = bidy_offset + by;
         int g_x = bidx_offset + bx;
         // Load prey counts into shared memory
         s_target_count[by*blockDim.x + bx] = prey_cell_count[g_y*d_grid_width + g_x];
         // Wait till all threads complete
         __syncthreads();
         // If an agent is occupying this cell and is searching
         if(pred_s >= 0){
```

```
            for(int s_y = 0, yy = bidy_offset; s_y < blockDim.y; s_y++, yy++){
               for(int s_x = 0, xx = bidx_offset; s_x < blockDim.x; s_x++, xx++){
                  // Offset from current location
                  int dy = yy - y;
                  int dx = xx - x;
                  float distance = (dy*dy) + (dx*dx);
                  // Check if offset coordinate is in vision
                  if( distance <= (vision_radius*vision_radius) ){
                     // Do not process neighbouring cells
                     if( (dy >= -1 && dy <= 1) && (dx >= -1 && dx <= 1)){
                        continue;
                     }
                     // Get current target count from shared memory
                     int curr_prey_count = s_target_count[s_y*blockDim.x + s_x];
                     // Continue if no agents here
                     if(curr_prey_count <= 0){
                        continue;
                     }
                     // Index into move direction most agent array
                     int d_g_n_d_index = -1;
                     // Check which direction this cell count belongs to by comparing
                     if((dx < 0) && (dy < 0)){
                        d_g_n_d_index = 5;
                     } // top index 2
                     else if((dx == 0) && (dy < 0)){
                        d_g_n_d_index = 2;
                     } // top right index 7
                     ... // Other direction
                     // Calculate n_agents / distance
                     double curr_ni_di = curr_prey_count / sqrt(distance);
                     // Accumulate sum_ni_di
                     sum_ni_di += curr_ni_di;
                     // Accumulate direction group
                     direction_group_most_agents[d_g_n_d_index] += curr_ni_di;
                  }
               }
            }
         }
         // Wait till all threads have finished using the shared memory
         __syncthreads();
      }
   }
   // Only process if there is an agent
   if(pred_s >= 0){
      get_agent_direction_indx_prob(..., ..., ..., ..., ...); // Set direction
   }
}

// This will write a move direction based on probability and a random decimal.
__device__ void get_agent_direction_indx_prob(double *direction_group_most_agents, int &
    move_direction_index, double sum_ni_di, int tid, curandState *rng_state){
   double upper = 0.0;
   // Random uniform double to choose a direction
   double random_uniform = curand_uniform_double(&rng_state[tid]);
   for(int i = 1; i < 9; i++){
      double prob = direction_group_most_agents[i] / sum_ni_di;
      double lower = upper;
      upper += prob;
      if(random_uniform > lower && random_uniform <= upper){
         move_direction_index = i;
         break;
      }
   }
}
```

Bibliography

[1] Christopher G Langton et al. Artificial life. 1989.

[2] Christopher G Langton. Studying artificial life with cellular automata. 1986.

[3] Rodney A Brooks. Artificial life and real robots. In *Proceedings of the First European Conference on artificial life*, pages 3–10, 1992.

[4] Christopher G. Langton. *Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[5] Jacques de Vaucanson. *An Account of the Mechanism of an Automaton, Or Image Playing on the German-Flute*. T. Parker, and sold by Mr. Stephen Varillon at the Long Room, at the Opera, 1742.

[6] Jessica Riskin. The defecating duck, or, the ambiguous origins of artificial life. *Critical Inquiry*, 29(4):599–633, 2003.

[7] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[8] Claude E Shannon. A universal turing machine with two internal states. *Automata studies*, 34:157–165, 1956.

[9] William Aspray. The origins of john von neumann's theory of automata. *Glimm et al.[GIS90]*, pages 289–309, 1990.

[10] John Von Neumann, Arthur W Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.

[11] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game life. *Scientific American*, 223(4):120–123, 1970.

[12] Andrew Adamatzky. *Game of life cellular automata*, volume 1. Springer, 2010.

[13] Christopher G Langton. Self-reproduction in cellular automata. 1984.

[14] Luc Steels and Rodney Brooks. *The artificial life route to artificial intelligence: Building embodied, situated agents*. Routledge, 2018.

[15] Luc Steels. *The biology and technology of intelligent autonomous agents*, volume 144. Springer Science & Business Media, 2012.

[16] David McFarland. Animals as cost-based robots. *International Studies in the Philosophy of Science*, 6(2):133–153, 1992.

[17] Luc Steels. Cooperation between distributed agents through self-organisation. In *Intelligent Robots and Systems' 90.'Towards a New Frontier of Applications', Proceedings. IROS'90. IEEE International Workshop on*, pages 8–14. IEEE, 1990.

[18] Christoph Adami. *Introduction to Artificial Life*. Springer-Verlag, 1998. ISBN 0-387-94646-2.

[19] Sol Spiegelman. An approach to the experimental analysis of precellular evolution. *Quarterly reviews of biophysics*, 4(2-3):213–253, 1971.

[20] Andrew Adamatzky and Maciej Komosinski. *Artificial life models in software*. Springer, 2005.

[21] Chris Adami, C Titus Brown, and W Kellogg. Evolutionary learning in the 2d artificial life system avida. In *Artificial life IV*, volume 1194, pages 377–381. The MIT Press Cambridge, MA, 1994.

[22] Thomas S Ray. Evolution and optimization of digital organisms. *Scientific Excellence in Supercomputing: The IBM 1990 Contest Prize Papers*, 1991.

[23] Charles Ofria and Claus O Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial life*, 10(2):191–229, 2004.

[24] TS Ray. Overview of tierra at atr. *Technical Information*, (15), 2001.

[25] Maciej Komosiński and Szymon Ulatowski. Framsticks: Towards a simulation of a nature-like world, creatures and evolution. In *European Conference on Artificial Life*, pages 261–265. Springer, 1999.

[26] Nick Collier. Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36:2003, 2003.

[27] Mitchel Resnick. Starlogo: an environment for decentralized modeling and decentralized thinking. In *Conference companion on Human factors in computing systems*, pages 11–12. ACM, 1996.

[28] John Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[29] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH computer graphics*, volume 21, pages 25–34. ACM, 1987.

[30] Stewart W Wilson. Knowledge growth in an artificial animal. In *Adaptive and Learning Systems*, pages 255–264. Springer, 1986.

[31] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[32] Stewart W Wilson. The animat path to ai. 1991.

[33] Agnès Guillot and Jean-Arcady Meyer. The animat contribution to cognitive systems research. *Cognitive Systems Research*, 2(2):157–165, 2001.

[34] Vladimir G. Red'ko, Konstantin V. Anokhin, Mikhail S. Burtsev, Alexander I. Manolov, Oleg P. Mosalov, Valentin A. Nepomnyashchikh, and Danil V. Prokhorov. Project "animat brain": Designing the animat control system on the basis of the functional systems theory. In *SAB ABiALS*, 2006.

[35] Richard S Sutton. Reinforcement learning architectures for animats. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 288–296, 1991.

[36] Colin Angle. *Genghis, a six legged autonomous walking robot*. PhD thesis, Massachusetts Institute of Technology, 1989.

[37] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviors. In *AAAI*, 1990.

[38] David Ball, Scott Heath, Michael Milford, Gordon Wyeth, and Janet Wiles. A navigating rat animat. In *ALIFE*, 2010.

[39] Siyuan Feng, Eric Whitman, X Xinjilefu, and Christopher G Atkeson. Optimization based full body control for the atlas robot. In *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, pages 120–127. IEEE, 2014.

[40] Christoph Adami. *Introduction to artificial life*, volume 1. Springer Science & Business Media, 1998.

[41] David P Bartel and Jack W Szostak. Isolation of new ribozymes from a large pool of random sequences [see comment]. *Science*, 261(5127):1411–1418, 1993.

[42] Martin C Wright and Gerald F Joyce. Continuous in vitro evolution of catalytic function. *Science*, 276(5312):614–617, 1997.

[43] Simon Taylor. *Agent-based modeling and simulation*. Springer, 2014.

[44] Charles Macal and Michael North. Introductory tutorial: Agent-based modeling and simulation. In *Proceedings of the 2014 winter simulation conference*, pages 6–20. IEEE Press, 2014.

[45] John H Miller and Scott E Page. *Complex adaptive systems: An introduction to computational models of social life*, volume 17. Princeton university press, 2009.

[46] John H Holland. Studying complex adaptive systems. *Journal of Systems Science and Complexity*, 19(1):1–8, 2006.

[47] Thomas C Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.

[48] Robert Axelrod. Effective choice in the prisoner's dilemma. *Journal of conflict resolution*, 24(1):3–25, 1980.

[49] Robert Axelrod. More effective choice in the prisoner's dilemma. *Journal of Conflict Resolution*, 24(3):379–403, 1980.

[50] Joshua M Epstein and Robert Axtell. *Growing artificial societies: social science from the bottom up*. Brookings Institution Press, 1996.

[51] Robert Axelrod. An agent-based model of social influence: Local convergence and global polarization. Technical report, Working Paper 95-03-028, Santa Fe Institute, Santa Fe, NM, 1995.

[52] Robert Axelrod et al. Resources for agent-based modeling.

[53] K. A. Hawick, H. A. James, and C. J. Scogings. A zoology of emergent patterns in a predator-prey simulation model. In H. Nyongesa, editor, *Proceedings of the Sixth IASTED International Conference on Modelling, Simulation, and Optimization*, pages 84–89, Gabarone, Botswana, September 2006.

[54] Chris Scogings. Towards an agent-based simulation of predators developing a search image. In *Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV)*, pages 55–61, Las Vegas, July 2015. The Steering Committee of The World Congress in Computer Science, (WorldComp).

[55] KJ Mock and JW Testa. An agent-based model of predator-prey relationships between transient killer whales and other marine mammals. *University of Alaska Anchorage, Anchorage, AK, Tech. Rep*, 2007.

[56] J Ward Testa, Kenrick J Mock, Cameron Taylor, Heather Koyuk, Jessica R Coyle, and Russell Waggoner. Agent-based modeling of the dynamics of mammal-eating killer whales and their prey. *Marine Ecology Progress Series*, 466:275–291, 2012.

[57] Colin M Henein and Tony White. Agent-based modelling of forces in crowds. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 173–184. Springer, 2004.

[58] Ameya Shendarkar, Karthik Vasudevan, Seungho Lee, and Young-Jun Son. Crowd simulation for emergency response using bdi agent based on virtual reality. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 545–553. IEEE, 2006.

[59] Neal Wagner and Vikas Agrawal. An agent-based simulation system for concert venue crowd evacuation modeling in the presence of a fire disaster. *Expert Systems with Applications*, 41(6):2807–2815, 2014.

[60] Thomas E Gorochowski. Agent-based modelling in synthetic biology. *Essays in biochemistry*, 60(4):325–336, 2016.

[61] Victor Garcia, Mirko Birbaumer, and Frank Schweitzer. Testing an agent-based model of bacterial cell motility: How nutrient concentration affects speed distribution. *The European Physical Journal B*, 82(3-4):235, 2011.

[62] Forrest Stonedahl and Uri Wilensky. Finding forms of flocking: Evolutionary search in abm parameter-spaces. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 61–75. Springer, 2010.

[63] Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of simulation*, 4(3):151–162, 2010.

[64] Nicholas R Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):277–296, 2000.

[65] Charles M Macal and Michael J North. Tutorial on agent-based modeling and simulation. In *Simulation conference, 2005 proceedings of the winter*, pages 14–pp. IEEE, 2005.

[66] C. J. Scogings and K. A. Hawick. Altruism amongst spatial predator-prey animats. In S. Bullock, J. Noble, R. Watson, and M. Bedau, editors, *Proc. 11th Int. Conf. on the Simulation and Synthesis of Living Systems (ALife XI)*, pages 537–544, Winchester, UK, 5-8 August 2008. MIT Press.

[67] Alessandro Troisi, Vance Wong, and Mark A. Ratner. An agent-based approach for modeling molecular self-organization. *Proceedings of the National Academy of Sciences*, 102(2):255–260, 2005.

[68] Amgad Naiem, M Reda, Mohammed El-Beltagy, and I El-Khodary. An agent based approach for modeling traffic flow. pages 1 – 6, 04 2010.

[69] C Bongiorno, S Miccichè, R Mantegna, G Gurtner, F Lillo, L Valori, M Ducci, B Monechi, and S Pozzi. An agent based model of air traffic management. In *Third SESAR Innovation Days*. Dirk Schaefer, 2013.

[70] GN Gilbert and L Hamill. Social circles: A simple structure for agent-based social network models. *Journal of Artificial Societies and Social Simulation*, 12(2), 2009.

[71] Andrew Crooks. *Agent-based Models and Geographical Information Systems*, pages 63–77. 01 2015.

[72] Christopher Bone and Mark Altaweel. Modeling micro-scale ecological processes and emergent patterns of mountain pine beetle epidemics. *Ecological modelling*, 289:45–58, 2014.

[73] Aurélien Vermeir and Hugues Bersini. Best practices in programming agent-based models in economics and finance. In *Advances in Artificial Economics*, pages 57–68. Springer, 2015.

[74] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 swarmfest workshop*, volume 8, pages 316–327. Michigan, USA, 2004.

[75] David Hiebeler et al. The swarm simulation system and individual-based modeling. Citeseer, 1994.

[76] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA, 2004.

[77] Elizabeth Sklar. Netlogo, a multi-agent simulation environment, 2007.

[78] 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.

[79] Andrew Adamatzky. Framsticks. *Kybernetes*, 29(9/10), 2000.

[80] Yuri G Karpov. Anylogic: A new generation professional simulation tool. In *VI International Congress on Mathematical Modeling, Nizni-Novgorog, Russia*, 2004.

[81] Steven F Railsback, Steven L Lytinen, and Stephen K Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623, 2006.

[82] Alan G Isaac. The abm template models: A reformulation with reference implementations. *Journal of Artificial Societies and Social Simulation*, 14(2):5, 2011.

[83] Nicolas Bacaër. Verhulst and the logistic equation (1838). In *A Short History of Mathematical Population Dynamics*, pages 35–39. Springer, 2011.

[84] Thomas Robert Malthus. *An Essay on the Principle of Population..* 1872.

[85] Alfred J Lotka. Contribution to the theory of periodic reactions. *The Journal of Physical Chemistry*, 14(3):271–274, 1910.

[86] Alfred J Lotka. Elements of physical biology. *Science Progress in the Twentieth Century (1919-1933)*, 21(82):341–343, 1926.

[87] Vito Volterra. Fluctuations in the abundance of a species considered mathematically, 1926.

[88] Alan A Berryman. The orgins and evolution of predator-prey theory. *Ecology*, 73(5):1530–1535, 1992.

[89] Crawford Stanley Holling. The functional response of invertebrate predators to prey density. *The Memoirs of the Entomological Society of Canada*, 98(S48):5–86, 1966.

[90] Patrick H Leslie. Some further notes on the use of matrices in population mathematics. *Biometrika*, 35(3/4):213–245, 1948.

[91] ME Solomon. The natural control of animal populations. *The Journal of Animal Ecology*, pages 1–35, 1949.

[92] Roger Arditi and Lev R Ginzburg. Coupling in predator-prey dynamics: ratio-dependence. *Journal of theoretical biology*, 139(3):311–326, 1989.

[93] Nicolas Bacaër. Lotka, volterra and the predator–prey system (1920–1926). In *A short history of mathematical population dynamics*, pages 71–76. Springer, 2011.

[94] L. V. Nedorezov. The dynamics of the lynx–hare system: an application of the lotka–volterra model. *Biophysics*, 61(1):149–154, Jan 2016.

[95] Vincenzo Capasso and Gabriella Serio. A generalization of the kermack-mckendrick deterministic epidemic model. *Mathematical Biosciences*, 42(1-2):43–61, 1978.

[96] James H Espenson. *Chemical kinetics and reaction mechanisms*, volume 102. Citeseer, 1995.

[97] Giancarlo Gandolfo. The lotka-volterra equations in economics: An italian precursor. *Economia Politica*, XXIV:343–348, 12 2007.

[98] K. A. Hawick. Spectral analysis of growth in spatial lotka-volterra models. In *Proc. International Conference on Modelling and Simulation*, number 685-030, pages 14–20, Gabarone, Botswana, 6-8 September 2010. IASTED.

[99] K. A. Hawick, D. P. Playne, and C. J. Scogings. Simulating the generalised spatial lotka-volterra equations with multiple species on gpus with automatic code generation. In *Proc. 12th IASTED Int. Conf. on Parallel and Distributed Computing and Networks (PDCN'13)*, Innsbruck, Austria, 11-13 February 2013. IASTED.

[100] Dara Quach, Daniel Playne, and Ken Hawick. Simulations of complex feeding chains in the lotka-volterra predator-prey model. *Proceedings of the IASTED International Conference on Modelling, Identification and Control*, 03 2014.

[101] DQ Quach, JM Willemse, V Du Preez, and KA Hawick. Species survivability and altitude dependence in a lotka-volterra predator-prey spatial-agent based system. In *Proceedings of the International Conference on Bioinformatics & Computational Biology (BIOCOMP)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer , 2013.

[102] Hugo Thierry, David Sheeren, Nicolas Marilleau, Nathalie Corson, Marion Amalric, and Claude Monteil. From the lotka–volterra model to a spatialised population-driven individual-based model. *Ecological modelling*, 306:287–293, 2015.

[103] Huaying Wang, Yulei Pang, Luona Zhang, and Clyde F Martin. A predator–prey interaction from micro to macro perspectives. *Nonlinear Dynamics*, 79(3):2017–2025, 2015.

[104] JHP Dawes and MO Souza. A derivation of holling's type i, ii and iii functional responses in predator–prey systems. *Journal of theoretical biology*, 327:11–22, 2013.

[105] WJ Chivers. A comparison of indivdual-based and dynamic modelling using bullant.

[106] California Institute of Technology. Digital life laboratory. Available at https://authors.library.caltech.edu/13705/.

[107] Thomas S Ray. Evolution, ecology and optimization of digital organisms. Technical report, Citeseer, 1992.

[108] H. A. James, C. J. Scogings, and K. A. Hawick. A framework and simulation engine for studying artificial life. *Research Letters in the Information and Mathematical Sciences*, 6(ISSN 1175-2777):143–155, May 2004.

[109] K. A. Hawick, C. J. Scogings, and H. A. James. Defensive spiral emergence in a predator-prey model. *Complexity International*, 12(msid37):1–10, October 2008. ISSN 1320-0682.

[110] K. A. Hawick, C. J. Scogings, and H. A. James. Spatial emergence of genotypical tribes in an animat simulation model. In S. G. Henderson, B. Biller, M. H. Hsieh, J. D. Tew, and R. R. Barton, editors, *Proc. 2007 Winter Simulation Conference 2007 (WSC2007)*, pages 1216–1222, Washington DC, USA, 9-12 December 2007. ISBN 1-4244-1306-0.

[111] K.A. Hawick and C. J. Scogings. Resource scarcity effects on spatial species distribution in animat agent models. In K. Grigoriadis, editor, *Proc. IASTED International Conference on Environmental Modelling and Simulation, 16-18 November, Orlando, USA*, pages 284–289, Orlando, USA., 16-18 November 2008. IASTED. ISBN 978-0-88986-777-2.

[112] K. A. Hawick and C. J. Scogings. Emergent spatial agent segregation. In *Proc 2008 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 34–40, Sydney, Australia, 8-9 December 2008. IEEE/WIC/ACM. ISBN 978-0-7695-3496-1.

[113] K. A. Hawick and C. J. Scogings. Spatial pattern growth and emergent animat segregation. *Web Intelligence and Agent Systems*, 8(2):165–179, 2010. ISSN 1570-1263.

[114] K. A. Hawick and C. J. Scogings. Hierarchical relationships and spatial emergence amongst multi-species animats. In *Proc. IASTED International Symposium on Modelling and Simulation (MS 2009)*, number 670-036, pages 1–6, Banff, Alberta, Canada, 6-8 July 2009. IASTED. CSTN-045.

[115] C. J. Scogings and K. A. Hawick. Modelling predator camouflage behaviour and tradeoffs in an agent-based animat model. In *Proc. IASTED International Conference on Modelling and Simulation (MS'2013)*, number CSTN-184, pages 802–032, Banff, Canada, 17-19 July 2013. WorldComp.

[116] C. J. Scogings and K. A. Hawick. Emergent system effects from microscopic evasion choices in a predator-prey simulation. In *Proc. 10th International Conference on Genetic and Evolutionary Methods (GEM'13)*, number CSTN-188, page GEM3895, Las Vegas, USA, 22-25 July 2013. WorldComp.

[117] C. J. Scogings and K. A. Hawick. Intelligent and adaptive animat resource trading. In *Proc. 2009 International Conference on Artificial Intelligence (ICAI 09)*, pages 85–90, Las Vegas, USA, 13-16 July 2009. WorldComp.

[118] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[119] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[120] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[121] Ted Sanders. The moore's law of moore's laws. *MRS Bulletin*, 40(11):991–992, 2015.

[122] Andrew A Chien and Vijay Karamcheti. Moore's law: The first ending and a new beginning. *Computer*, 46(12):48–53, 2013.

[123] Kenneth Flamm. Measuring moore's law: Evidence from price, cost, and quality indexes. Technical report, National Bureau of Economic Research, 2018.

[124] Michael J Flynn. Computer engineering 30 years after the ibm model 91. *Computer*, 31(4):27–31, 1998.

[125] John Cocke and D Slotnick. The use of parallelism in numerical calculations. *IBM RC-55*, 1958.

[126] James P Anderson, Samuel A Hoffman, Joseph Shifman, and Robert J Williams. D825-a multiple-computer system for command & control. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, pages 86–96. ACM, 1962.

[127] George H Barnes, Richard M Brown, Maso Kato, David J Kuck, Daniel L Slotnick, and Richard A Stokes. The illiac iv computer. *IEEE Transactions on computers*, 100(8):746–757, 1968.

[128] Simson Garfinkel. *Architects of the information society: 35 years of the Laboratory for Computer Science at MIT*. MIT press, 1999.

[129] Gregory V Wilson. *A Chronology of Major Events in Parallel Computing*. University of Toronto. Computer Systems Research Institute, 1994.

[130] William A Wulf and C Gordon Bell. C. mmp: a multi-mini-processor. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, pages 765–777. ACM, 1972.

[131] Gregory V Wilson. The history of the development of parallel computing. *URL: http://ei. cs. vt. edu/history/Parallel. html*, 1994.

[132] Harold S Stone. Parallel processing with the perfect shuffle. *IEEE transactions on computers*, 100(2):153–161, 1971.

[133] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[134] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[135] 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.

[136] DP Playne. *Generative programming methods for parallel partial differential field equation solvers*. PhD thesis, Ph. D. thesis, Computer Science, Massey University, 2011.

[137] Guillem Pratx and Lei Xing. Gpu computing in medical physics: A review. *Medical physics*, 38(5):2685–2697, 2011.

[138] Harald Vranken. Sustainability of bitcoin and blockchains. *Current opinion in environmental sustainability*, 28:1–9, 2017.

[139] Karl J O'Dwyer and David Malone. Bitcoin mining and its energy footprint. 2014.

[140] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz. Comparative performance analysis of intel (r) xeon phi (tm), gpu, and cpu: a case study from microscopy image analysis. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1063–1072. IEEE, 2014.

[141] TOP500.org. TOP 500 Supercomputer Sites. `https://nvlabs.github.io/cub/structcub_1_1_device_select.html`. Last accessed March 2019.

[142] Charles I Saidu, AA Obiniyi, and Peter O Ogedebe. Overview of trends leading to parallel computing and parallel programming. *British Journal of Mathematics & Computer Science*, 7(1):40, 2015.

[143] Arno Leist, Daniel Peter Playne, and Kenneth A. Hawick. Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience*, 21:2400–2437, 2009.

[144] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[145] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[146] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[147] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 362–370. ACM, 2010.

[148] Ramtin Shams, Parastoo Sadeghi, Rodney A Kennedy, and Richard I Hartley. A survey of medical image registration on multicore and the gpu. *IEEE Signal Processing Magazine*, 27(2):50–60, 2010.

[149] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. Using openmp vs. threading building blocks for medical imaging on multi-cores. In *European Conference on Parallel Processing*, pages 654–665. Springer, 2009.

[150] Mikhail Smelyanskiy, David Holmes, Jatin Chhugani, Alan Larson, Douglas M Carmean, Dennis Hanson, Pradeep Dubey, Kurt Augustine, Daehyun Kim, Alan Kyker, et al. Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures. *IEEE transactions on visualization and computer graphics*, 15(6):1563–1570, 2009.

[151] Jega Anish Dev. Bitcoin mining acceleration and performance quantification. In *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, pages 1–6. IEEE, 2014.

[152] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838. IEEE, 2008.

[153] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[154] Daniel Strigl, Klaus Kofler, and Stefan Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 317–324. IEEE, 2010.

[155] Andreas Horni, Kai Nagel, and Kay W Axhausen. *The multi-agent transport simulation MATSim*. Ubiquity Press London, 2016.

[156] Nikolaos Bezirgiannis, ISWB Prasetya, and Ilias Sakellariou. Hlogo: A parallel haskell variant of netlogo. In *2016 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, pages 1–10. IEEE, 2016.

[157] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cordasco, and Vittorio Scarano. Openabl: A domain-specific language for parallel and distributed agent-based simulations. In *European Conference on Parallel Processing*, pages 505–518. Springer, 2018.

[158] Néstor Ferrando, MA Gosalvez, Joaquín Cerdá, R Gadea, and Kazuo Sato. Octree-based, gpu implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. *Computer Physics Communications*, 182(3):628–640, 2011.

[159] Xiaosong Li, Wentong Cai, and Stephen John Turner. Supporting efficient execution of continuous space agent-based simulation on gpu. *Concurrency and Computation: Practice and Experience*, 28(12):3313–3332, 2016.

[160] Melissa Tracy, Magdalena Cerdá, and Katherine M Keyes. Agent-based modeling in public health: current applications and future directions. *Annual review of public health*, 39:77–94, 2018.

[161] Elizaveta Kosiachenko. *Efficient GPU Parallelization of the Agent-Based Models Using MASS CUDA Library*. PhD thesis, 2018.

[162] Mikola Lysenko, Roshan M DSouza, et al. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[163] 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*, pages 129–132. ACM, 2018.

[164] Paul Richmond. Resolving conflicts between multiple competing agents in parallel simulations. In *European Conference on Parallel Processing*, pages 383–394. Springer, 2014.

[165] Fabien Michel, Grégory Beurier, and Jacques Ferber. The turtlekit simulation platform: Application to complex systems. In *SITIS: Signal-Image Technology and Internet–Based Systems*, 2005.

[166] Guillaume Laville, Christophe Lang, Bénédicte Herrmann, Laurent Philippe, Kamel Mazouzi, and Nicolas Marilleau. Mcmas: A toolkit for developing agent-based simulations on many-core architectures. *Multiagent and Grid Systems*, 11(1):15–31, 2015.

[167] Paul Richmond, Simon Coakley, and Daniela M Romano. A high performance agent based modelling framework on graphics card hardware with cuda. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 1125–1126. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[168] C. Adami. On modeling life. In R. Brooks and P. Maes, editors, *Proc. Artificial Life IV*, pages 269–274. MIT Press, 1994.

[169] T.S. Ray. An approach to the synthesis of life. *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*, xi:371–408, 1991.

[170] Toby Tyrrell and John E. W. Mayhew. Computer simulation of an animal environment. In Jean-Arcady Meyer and Stewart W. Wilson, editors, *From Animals to Animats, Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 263–272, 1991.

[171] Larry Yaeger. Computational genetice, physiology. metabolism, neural systems, learning, vision and behavior or polyworld: Life in a new context. In Chris Langton, editor, *Proc Artificial Life III Conference*, 1994.

[172] Jon M Watts. Animats: computer-simulated animals in behavioral research. *Journal of animal science*, 76(10):2596–2604, 1998.

[173] J. H. Holland. Echoing emergence: Objectives, rough definitions, and speculations for echo-class models. In G. A. Cowan, D. Pines, and D. Meltzer, editors, *Complexity: Metaphors, Models and Reality*, pages 309–342. Addison-Wesley, Reading, MA, 1994.

[174] John H Holland. *Emergence: From chaos to order*. OUP Oxford, 2000.

[175] Mark A Bedau. Philosophical aspects of artificial life. 1996.

[176] K.A. Hawick and C. J. Scogings. Animat swarms and their role in arterial blockage phenomena. In *Proc. 2009 International Conference on Genetic and Evolutionary Methods (GEM 09)*, pages 112–117, Las Vegas, USA, 13-16 July 2009. WorldComp.

[177] C. J. Scogings and K. A. Hawick. An Agent-Based Model of the Battle of Isandlwana. In *Proc. 2012 Winter Simulation Conference*, number CSTN-116, Berlin, Germany, 9-12 December 2012. WSC. ISBN: 978-1-4673-4780-8.

[178] H. A. James, C. J. Scogings, and K. A. Hawick. Parallel synchronization issues in simulating artifical life. In Teofilo Gonzalez, editor, *Proc. 16th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS)*, pages 815–820, Cambridge, MA, USA, 9-11 November 2004. IASTED. ISSN 1925-7937; ISBN 0-88986-421-7.

[179] DQ Quach, DP Playne, and CJ Scogings. The effect of changing search patterns in an agent-based model. In *Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV)*, page 81. The Steering Committee of The World Congress in Computer Science, 2016.

[180] K. A. Hawick, H. A. James, and C. J. Scogings. A virtual prolog approach to implementing beliefs, desires and intentions in animat agents. In Shichao Zhang and Ray Jarvis, editors, *Proc 18th Australian Joint Conference on Advances in Artificial Intelligence (AI'05)*, number LNAI 3809 in CSTN-022, Sydney, Australia, 5-8 December 2005. Springer. ISSN 0302-9743, ISBN 3-540-30462-2.

[181] K. A. Hawick, H. A. James, and C. J. Scogings. Non-monotonic phase transition edges in the spatial prisoners' dilemma. In *Proc IASTED Int. Conf on Applied Simulation and Modelling (ASM 2007)*, number 581-095, pages 213–218, Palma de Mallorca, Spain, 29-31 August 2007. IASTED.

[182] C. J. Scogings, K. A. Hawick, and H. A. James. Boundary conditions and locality in an agent-based predator-prey model. In *Proc. 17th IASTED Int. Conf on Applied Simulation and Modelling*, number 609-016, pages 1–6, Corfu, Greece, 23-25 June 2008. IASTED.

[183] C. J. Scogings and K. A. Hawick. Optimal data structures for spatially localised agent-based automata and hybrid systems. In *Proc. Int. Conf. on Artificial Intelligence and Soft Computing*, pages 221–227, Napoli, Italy, 25-27 June 2012. IASTED.

[184] Norman Margolus and Tommaso Toffoli. Cellular automata machines. *Complex Systems*, 1:967–993, 1987.

[185] Stephen Wolfram. *Theory and Applications of Cellular Automata*. World Scientific, 1986.

[186] Murray Eden. A two-dimensional growth process. In *Proc. Fourth Berkeley Symposium on Mathematics, Statistics and Probability*, volume 4, pages 223–239, Berkeley, 1960. Univ. California Press.

[187] C. J. Scogings and K. A. Hawick. Multiphase updating - a practical approach to simulating animat agents. In *Proc. Int. Conf. on Artificial Intelligence (ICAI'12)*, pages 1–7, Las Vegas, USA, 16-19 July 2012. World-Comp.

[188] Donald Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 3rd edition, 1997.

[189] CPP Reference. C++17 random shuffle. `http://en.cppreference.com/w/cpp/algorithm/random_shuffle`, March 2018.

[190] C. J. Scogings, K. A. Hawick, and H. A. James. Tools and techniques for optimisation of microscopic artificial life simulation models. In H. Nyongesa, editor, *Proceedings of the Sixth IASTED International Conference on Modelling, Simulation, and Optimization*, pages 90–95, Gabarone, Botswana, 11-13 September 2006. IASTED.

[191] C. J. Scogings and K. A. Hawick. Global constraints and diffusion in a localised animat agent model. In *Proc. IASTED Int. Conf. on Applied Simulation and Modelling*, pages 14–19, Corfu, Greece, 23-25 June 2008. IASTED.

[192] Vitor Manuel Vieira Lópes. Parallelization strategies for spatial agent-based models. 2016.

[193] Bruce Merry. A performance comparison of sort and scan libraries for gpus. *Parallel Processing Letters*, 25(04):1550007, 2015.

[194] Ronald Aylmer Fisher, Frank Yates, et al. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, (Ed. 3.), 1949.

[195] Guojing Cong and David A Bader. An empirical analysis of parallel random permutation algorithms on smps. Technical report, Georgia Institute of Technology, 2006.

[196] NVIDIA CORPORATION. CUDA UnBound Documentation. `http://www.top500.org/`. Last accessed November 2018.

[197] K. A. Hawick, H. A. James, and C. J. Scogings. Roles of rule-priority evolution in animat models. In *Proc. Second Australian Conference on Artificial Life (ACAL 2005)*, pages 99–116, Sydney, Australia, 5-8 December 2005.

[198] Christian Collberg and Todd A Proebsting. Repeatability in computer systems research. *Communications of the ACM*, 59(3):62–69, 2016.

[199] Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[200] Hazel R Parry and Mike Bithell. Large scale agent-based modelling: A review and guidelines for model scaling. In *Agent-based models of geographical systems*, pages 271–308. Springer, 2012.

[201] N Fachada. *Agent-Based Modeling on High Performance Computing Architectures*. PhD thesis, INSTITUTO SUPERIOR TÉCNICO, 2016.

[202] NVIDIA. NVIDIA Turing GPU Architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018. Last accessed March 2019.

[203] NVIDIA. NVIDIA NVSWITCH The World's Highest-Bandwidth On-Node Switch. `http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf`, 2018. Last accessed March 2019.