The Impact of Minimising Data Movement on the Overall Performance of the Simulation of Complex Systems Applied to FLAME GPU

Eidah M. Alzahrani

The Department of Computer Science

Faculty of Engineering

University of Sheffield

A thesis submitted for the degree of Doctor of Philosophy

January 2021

I would like to dedicate this thesis to my beloved husband and my great parents for their love, support and encouragement...

Abstract

GPUs have been demonstrated to be highly effective at improving the performance of Multi-Agent Systems (MAS). One of the significant limitations of further performance improvements is in the memory bandwidth required to move agent data through the GPU's memory hierarchy. This thesis investigates the impact of data dependency on the FLAME GPU framework's overall performance as an example of Agent Based Modelling (ABM) platforms. This investigation includes discovering data dependency within FLAME GPU models. Two methods are proposed in order to minimise data movement during simulation using dependency information: (i) a functional method which is based on the concept of merging and splitting agent function; and (ii) data-aware method which uses of data dependency information to access a subset of agent and message memory at the variable level. This thesis also develops a method that allows automatic discovery of data dependencies from existing FLAME GPU models. This method is based on parsing an agent function file of a FLAME GPU model to extract all agent functions' data dependencies.

The scalability, computational complexity, internal memory requirements, and homogeneity of the agent and population of the model are examples of factors that may affect ABM applications' overall performance. This thesis presents a standard benchmark model designed to observe the system behaviour while testing these factors. An evaluation of the performance impact of minimising data movement has been carried out by implementing the proposed FLAME GPU methods using the benchmark model and the number of existing FLAME GPU models. The comparison between the current and new system shows that reducing data movement within a simulation improves overall performance.

Declaration

I hereby declare that I am the sole author of this thesis. The work presented in this thesis is original work and have not been submitted for any other degree or any other university. Parts of the work presented in Chapters 5, 7 and 8 have been published in a journal and conference proceedings as follows:

- Alzahrani, E., Richmond, P. and Simons, A.J., 2017, August. A formula-driven scalable benchmark model for ABM, applied to FLAME GPU. In European Conference on Parallel Processing (pp. 703-714). Springer, Cham.

- E. Alzahrani, A. J. H. Simons and P. Richmond, Data aware simulation of complex systems on GPUs. Proc. 17th. International Conference on High Performance Computing and Simulation (HPCS 2019), 15-19 July, Dublin, Ireland (2019). In: HPCS 2019, 17th International Conference on High Performance Computing and Simulation, eds. W. W. Smari, K. Zine-Dine (Piscataway NJ: IEEE, 2019), 567-574.

Acknowledgements

There are many people who deserve to be acknowledged in this thesis, and to whom I owe my sincere thanks:
To my supervisors Dr. Anthony Simons and Dr. Paul Richmond for their excellent supervision. This thesis would not have been possible without the immense support and patience offered by them. Their encouragement and advice have sustained me throughout every stage of my PhD journey.

To my internal PhD panel Dr. Siobhan North whose expertise no doubt guided me towards my chosen topic.
To my examiners Dr. Mike Stannett and Prof. Fiona Polack for valuable comments and useful discussion during the final viva of my defence.

To my colleagues in the Visual Computing Lab who have provided feedback on my work through informal discussion and research seminars.I would particularly like to thank Robert Chisholm, Peter Heywood, James Pyle who have always made themselves available to discuss issues regarding FLAME GPU and kindly provided much needed proofreading. Special thanks to Prof. Alcione de Paiva Oliveira for his support and help to understand FLAME GPU in the early stage of my PhD journey.

To Mozhgan, Nazrina, Lubna, Fatima, Mashael,Sadeen, Rabab and Najwa for their friendship and incredible support over the last few years.

To my dear parents Mohammad and shariffa for their prayers and keeping me strong throughout this journey. To my brothers and sisters for the love and support that I continuously receive.

To my husband Ahmed for unquestionable support throughout the more

difficult times of my PhD journey, and without his encouragement I would most likely have given up on my research long ago. To my beloved kids: Ghaida, Osama, Talah and Ali for their unconditional love and incredible support.

Finally, this work presented in this thesis would not have been possible without the financial support of AlBaha University and the Saudi Ministry of Education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Agent-based modelling (ABM) is a simulation technique used to simulate the actions and reactions of autonomous agents. Based on simple rules, these agents can communicate with each other and with their environment; this makes ABM a suitable approach for simulating complex systems. By using multi-core central processing units (CPUs), distributed systems and accelerators such as graphic processing units (GPUs), high performance computing (HPC) creates a suitable environment for handling complex operations. These operations may include, but are not limited to, big data, simulating complex systems, performing large-scale simulations, and conducting similarly extensive processes. Operating large-scale simulations by using agent-based systems has gained attention in many research areas, such as biology systems[61, 62, 129, 123, 112, 99], manufacturing systems[12], social studies[137], supply chains[85] and epidemiology systems such as those related to the COVID19 pandemic[136, 64, 161, 131, 135].

Parallelising such tasks within a simulation model can be achieved through different approaches depending on the architecture of the HPC computer, distributed clusters or even multi-core processors. However, the increase in performance achieved through CPU parallelism may be affected by a number of performance-limiting issues, including managing communication between dynamic resources (e.g. clusters of cores, often referred to as nodes), load balancing between cores and monitoring the state of a distributed simulation. Exploiting the shared memory parallel architecture of a GPU to run simulations has the potential to overcome many of these problems. CUDA technology facilitates direct access to GPU hardware by using a C-like language; however, achieving optimal performance using CUDA requires extensive knowledge of GPU architectures.

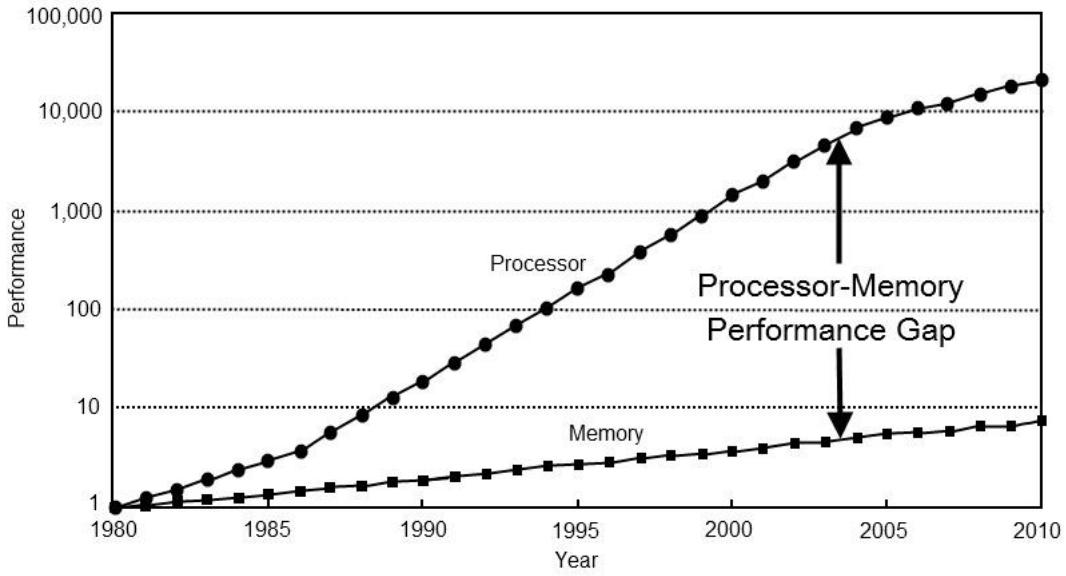

Figure 1.1: The gap between processor and memory performance over the years[1]

The Flexible Large-Scale Agent Modelling Environment for the GPU (FLAME GPU) framework allows agent-based modellers to run simulations on GPUs without an explicit understanding of GPU programming. Both the FLAME and FLAME GPU frameworks offer high efficiency in terms of the execution of running models under diverse architectures, such as multi-core processors and GPUs. The complexity of agent communication (which in FLAME involves indirect sending and receiving of messages) is similar to data transfers between different nodes in heterogeneous architectures. Agent transition functions are used to move an agent from one state to another by reading and writing the memories allocated to agent variables. In the case of data dependency between states, subsequent transition functions must wait for the new version of the memory to be updated. On the other hand, FLAME GPU is best suited for large populations with relatively simple memory requirements for each agent. This enables a suitable balance with the large number of threads required to hide memory access latencies with the limited register available. Thus, simulation is restricted to the memory space available on a single GPU device, as FLAME GPU lacks support for multiple GPU devices.

[1]Source: https://bitbashing.io/memory-performance.html

1.1 Motivation

The complexity and heterogeneous memory of HPC systems poses challenges to the minimization of the gap between processor speed and main memory cycle time. This gap doubles every one to two years, which makes it one of the most critical challenges in the computing industry according to Machanick [88], as shown in figure 1.1. Managing data movement between processors and memory (known as the memory wall problem) is the most obvious challenge, particularly in systems that deal with large amounts of heterogeneous data—for example, when simulating large models. Most simulators for ABM are therefore memory bound: the agent uses memory to hold its variables (or internal state) and communication between agents. The memory wall problem has become more evident, particularly in simulators using a streaming-based (data in, calculate, data out) approach to iteratively transform the memory of agents. This situation significantly impacts overall performance for large populations and scalable models since increasing amounts of data need to be moved.

Reducing data movement (data transfer between processors and system memory) in such systems will improve overall performance. A number of techniques (and associated studies) have focused on this goal, including load balancing [138, 66, 67], graph partitioning [13] [146], spatial partitioning (or spatial messaging) [129] [127] and others [134, 157].

To address this issue, this thesis proposes two approaches that can be applied to the simulation of complex multi-agent systems (MAS) on GPUs. We demonstrate a functional approach and a data-aware approach to simulation experimentally using FLAME GPU. However, the underlying data-aware approach abstraction is appropriate for any streaming-based MAS platform or model.

1.2 Research focus

The focus of this thesis is to investigate the impact of minimising data movement on FLAME GPU performance. The first stage will focus on designing and implementing a benchmark model that can be used to measure the performance of the current FLAME GPU. This model also will be used to evaluate the use of proposed approaches to enhance FLAME GPU performance. The concept of

this benchmark model can be implemented on other AMB platforms to collect benchmark data. In the second stage, this research will propose a new method that helps reduce the simulation execution time. The functional approach is based on merging some agents' functions after discovering data and message dependencies. The functional approach is particularly suitable for FLAME GPU. In the third stage, this thesis will focus on how the required data are extracted from agent behaviour during the simulation and how this information can be used to reduce data movement through what we term a data-aware approach. In this stage, the FLAME GPU software will be extended to demonstrate this technique. Finally, both approaches will be evaluated and validated in the last stage by using a benchmark model and other existing FLAME GPU models. A summary of the research focus is shown in figure 1.2

1.3 Thesis Aims

The aims of this thesis are as follows:

- Designing and implementing a general-purpose benchmark model using FLAME GPU. The fundamental concept of this model can be implemented in different ABM platforms to evaluate the modelling capabilities of these applications.
- Exploring the benefit of applying a functional approach to FLAME GPU through the new benchmark model. This approach is based on the concept of merging and splitting agent functions within FLAME GPU models.
- Developing a method that allows automatic discovery of data dependencies from existing CSmodels. This method is based on parsing an agent function file of a FLAME GPU model to extract all data dependencies between agent functions.
- Automating the processes of applying discounted variable level data within the automated process of generating efficient GPU code in FLAME GPU.

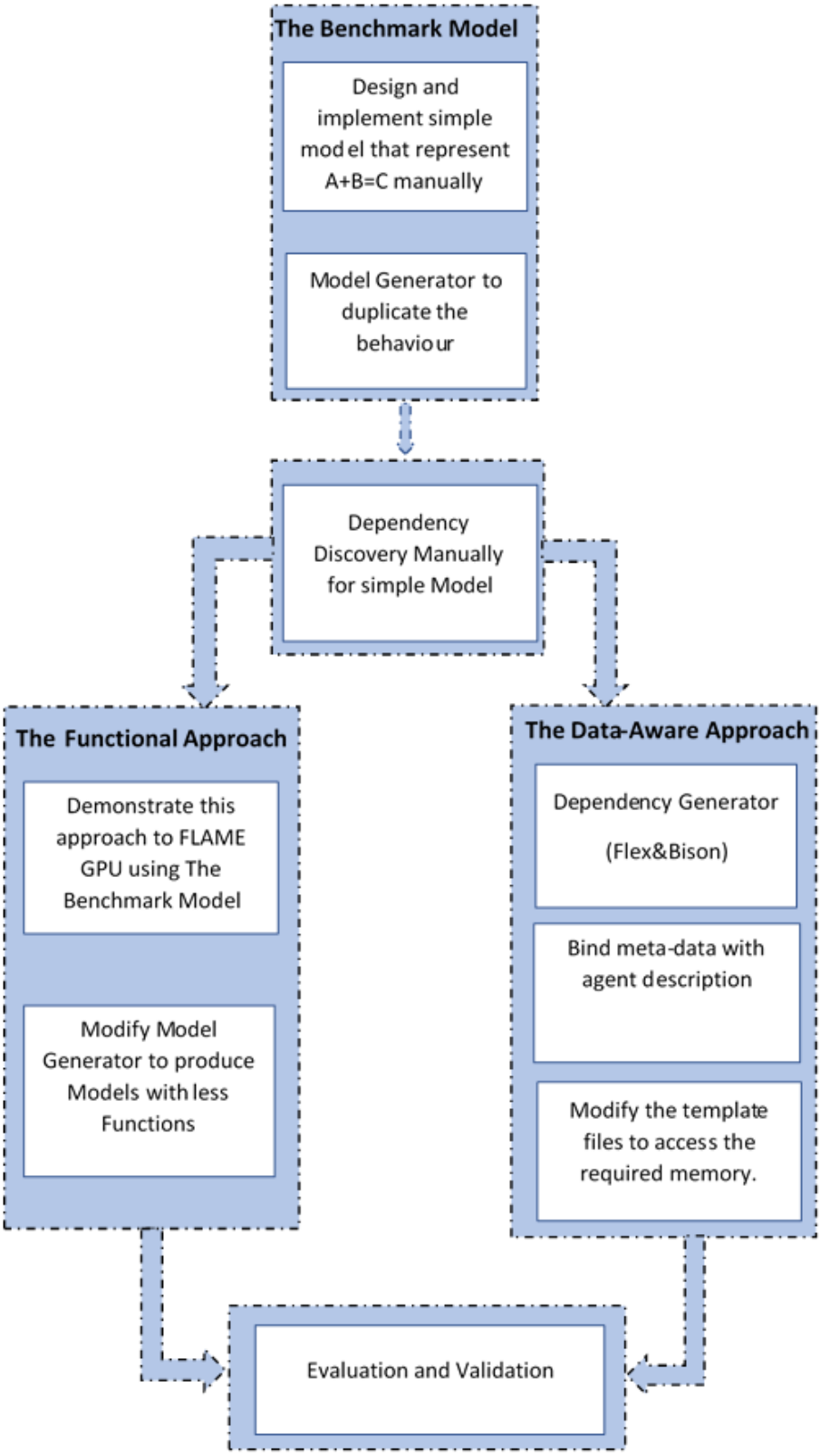

Figure 1.2: This is a summary of the research focus. It starts with designing and implementing the benchmark model. In the second stage, two approaches have been adopted to reduce memory access during the simulation applied to FLAME GPU. The final stage focuses on evaluation and validation of the proposed approaches.

1.4 Contribution of knowledge

The main contributions to knowledge are as follows:

1.4.1 A benchmark model

The growing number of frameworks for generating parallel ABMS applications has led to an increase in the number of studies evaluating the modelling capabilities of these applications. The main factor for benchmarking in most of these studies is based on observing system performance while increasing the population size. However, system scalability is not the only issue that may affect the overall performance of ABM applications. The first contribution of this thesis is a benchmark model that can be a standard for measuring the execution efficiency of the existing ABM systems. This benchmark model has been implemented for FLAME GPU, though the fundamental concept of this model can be implemented in different ABM platforms.

Our benchmark model can vary the following elements: system scalability, system homogeneity (for both agent complexity and population complexity) and the ability to handle an increase in the level of agent communication. These results offer insight into the performance characteristics of simulations and provide a baseline for measuring simulator improvements. This work was published in [8].

1.4.2 Data-aware approach

By adopting this approach, this thesis demonstrates that minimising data movement at the variable level within a complex system (CS) simulation improves overall performance. This contribution was achieved by automatically extracting the data dependencies of agents' functions and developing FLAME GPU to access the required agent and message memory during the simulation. The effects of this approach on performance are evaluated using the benchmark model and other existing CS models. This contribution was published in [9].

1.5 Outline of the thesis

The remainder of this thesis is presented in Chapters 2 to 9. These chapters are organised as follows.

- **Chapter 2: Background and Literature Review.** In this chapter, four topics are covered in more detail, including ABM and simulation, ABM on a GPU, and a brief introduction to FLAME GPU frameworks. This chapter also presents the impact of data dependencies on real-time high-performance computing by reviewing a number of studies and existing techniques for discovering data dependencies and reducing memory movement.

- **Chapter 3: Agent-Based Models for GPUs.** Here, the FLAME GPU platform is discussed in greater detail to provide the necessary context for the work in this thesis.

- **Chapter 4: Methods and Experimental Plan.** This chapter describes the methodologies used and the description of the tools required to perform both evaluation and validation of the proposed approaches

- **Chapter 5: Benchmarking Agent-Based Modelling Systems.** This chapter presents an overview of benchmarking ABM systems and reviews a number of tools that have been used for this purpose. New benchmark criteria are discussed within this chapter, as well as a new benchmark model that can help to measure the overall performance of ABM systems in different aspects. This new benchmark model is implemented in FLAME GPU and its evaluation and testing are discussed in this chapter.

- **Chapter 6: The Impact of Combining Agents' Functions on Overall Performance.** This chapter presents an experiment to discover the impact of reducing data movement within agent-based models. It is conducted through manual manipulation of existing models using a guided data-dependency process. The technique proposed combines and splits agent functions to reduce the amount of data movement during simulation. This chapter also discusses some examples of results after applying this approach to FLAME GPU using the benchmark model.

- **Chapter 7: A Data-Aware Model for Agent Representation.** This chapter explains the automation of discovering data dependencies and shows how the new discovery tool has been used and some examples of the results of its application. This chapter also presents a new form representation of an agent within X-machine models that simulates

data dependency at the variable level. After extracting data dependencies from existing models, the automated process of merging metadata with model specification files is described. The FLAME GPU software contains a number of XSLT templates, which are used to generate the dynamic GPU simulation code. This chapter describes modifying these templates to generate efficient simulation code that is data-aware and minimises data movement.

- **Chapter 8: Results.** This chapter evaluates the performance achieved by applying the data-aware discounted representation approach with the FLAME GPU software using a number of existing models. It also presents the results of comparing the simulation execution time of both systems (the current FLAME GPU and the extended FLAME GPU) to highlight the impact of variable level data dependencies on simulation performance.

- **Chapter 9: Conclusions.** This chapter concludes the thesis and highlights potential directions for future work.

Chapter 2

Background and Literature Review

2.1 Introduction

As outlined in the previous chapter, this thesis investigates the effects of minimising data movement on the overall performance of FLAME GPU. This chapter covers the background and reviews the literature of four key topics: ABM and simulation, ABM on the GPU, and FLAME GPU frameworks are included. The influence of data dependencies in real-time high-performance computing is also presented in this chapter.

2.2 Agent-Based Modelling and Simulation

In the last few years, the use of agent-based modelling and simulation (ABMS) has increased rapidly in various fields, such as ecology [51, 92], social science [70], economics [70, 104], computer science, businesses complexity [100] and earth science [89]. According to Bajracharya and Duboz [11], ABMS enables the construction of a complex system as a group of agents; all behaviours are described with respect to the individuals within this system. Grimm et al. also define the ABMS of complex systems as "dynamic networks of many interacting agents" [52] p.987. Agent-based platforms are a form of architecture that creates a suitable environment in which agents can exist and communicate to achieve the goal of the simulated model [11]. The most common use of ABMS in business is to support decision-making [86]. For example, it can be used to compare different marketing strategies to choose the best one or, in social

life,to optimise problems such as crowd behaviour and traffic flow. Depending on the number of agents, the agents' communication and the complexity of the model, modellers can build their systems on desktop computers, computing clusters or even HPCs using different ABMS toolkits [7].

This section will give a brief introduction to modelling and simulation systems and present a number of ABMS toolkits and platforms popular in different fields.

2.2.1 Early developments

ABMS initially came from AI and computer science, but it is now used for modelling artificial or non-artificial systems and is an independent research field. The earliest emergence of ABMS was in a device created by John Von Neumann and later termed cellular automata. The game of life model developed by Jon Conway in 1970 was the first simulation model to use cellular automata. This model is based on two states, alive and dead, where the cell state depends on the states of the neighbouring cells. In this game, Conway succeeded in creating a complex system using simple rules. The game of life model has a computational complexity as powerful as a Turing-complete computer, which can generate new objects, including copies of original agents. In the 1990s, ABMS developments continued to grow with the appearance of several tools in different fields, namely Swarm and NetLogo in the mid-1990s, Repast and MASON toolkits in the early 2000s and FLAME and FLAME GPU in the late 2000s.

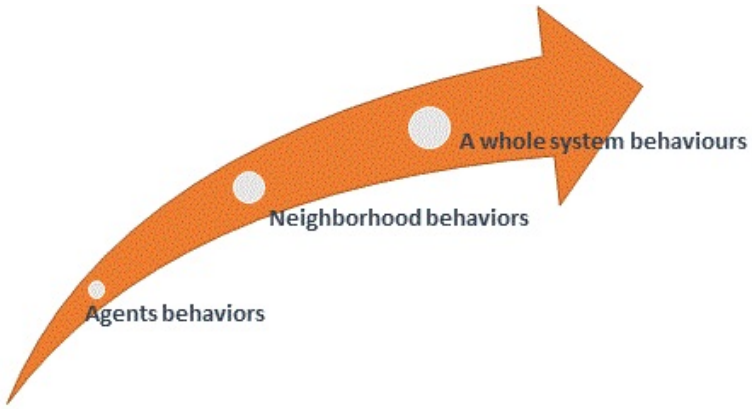

Figure 2.1: Bottom-up approach

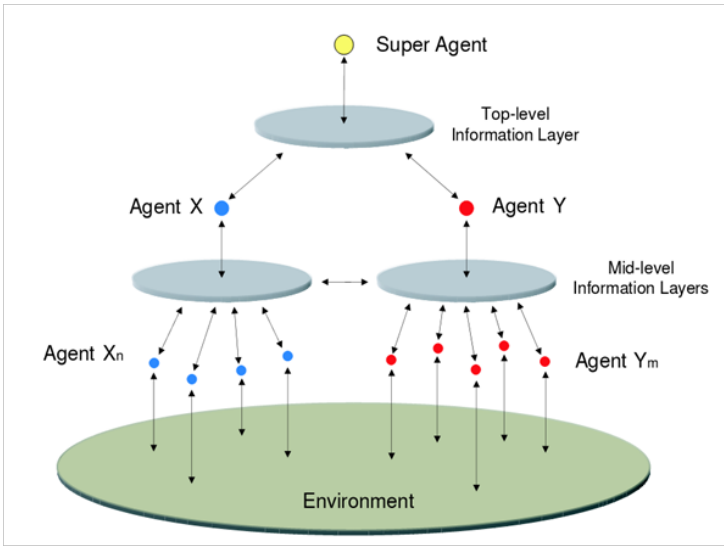

Figure 2.2: Top-down approach[1]

2.2.2 Basic concepts of ABMS

ABMS is a modelling approach that simulates the actions and reactions of individual entities and measures their effects on the whole system. Many phenomena, even complex ones, can be described as systems of autonomous agents following a number of rules to communicate with each other. ABMS follows a bottom-up approach, which means that each entity follows simple behaviour rules, as shown in figure 2.1. In contrast, a simulation following a top-down approach defines the behaviour of entities as a sequence of well-defined events, as shown in figure 2.2 In both approaches, the simulation results ultimately present the behaviours of the system as a whole, based on the behaviours of individual entities [162]. ABMS is relevant to many other fields, such as systems science, complexity science, systems dynamics, traditional modelling and simulation, and many branches of social sciences[86].

2.2.3 ABMS paradigms and methodologies

Two types of paradigms are used in ABMS:

- **Autonomous agent models** are used to study personal activities and behaviour. This paradigm can be used to model systems in non-computing-related scientific domains, such as life and ecological sciences [52]. In this type of model, an agent is self-directed and can communicate with its environment and other agents independently, such as cells in the genetic programs.

[1]The resource shown in Figure 2.2 is http://www.asdl.gatech.edu/INIT:AGENT.html

- **Agent-based models (system models)**, also known as multiagent systems, are used to study the actions and reactions of a collection of autonomous entities called agents. This type of ABMS is used to simulate dynamic complex systems and usually uses the same computational method as distributed artificial intelligence (DAI) systems. Computing scientific domains, such as AI and distributed autonomous systems, use this methodology in their applications [162]. Agent-based models follow a pattern-oriented modelling approach, which is used in bottom-up complex systems.

The most common paradigm used in ABM frameworks is object oriented programming (OOP). OOP is based on the concept of objects which provide a simple technique for creating models. Using features offered by the OOP methodology, including classes and methods, engineers can easily represent agents and agent behaviours as self-directed objects [7]. Most popular ABM frameworks are based on OOP concepts. Some of these also use Unified Modelling Language (UML) for system specification of high level agents. Some of these toolkits use Application Programming Interfaces (APIs) to help the user create model specifications. In these types of platforms, the API builds and describes the models, and the framework implements agent communication and agent behaviour scheduling [7]. The method of creating agents and defining agent specifications and agents' behaviour and attributes generally differ from one platform to another, depending on the type of ABM approach. In frameworks that use APIs, modellers can easily set up the input data visually using graphical user interfaces (GUIs) to create agents or by writing agent specifications in external files with customised file formats [87].

2.2.4 Agent-based modelling vs. equation-based modelling

In equation-based modelling (EBM), a system is simulated by executing a set of equations that represent the relationships between the system's variables. In contrast, agent-based modelling (ABM) simulates the internal behaviour of a collection of agents that combine to produce this high-level behaviour. Both approaches are used to build models that simulate real systems and solve complex issues and problems. However, they differ in how they execute the

models. Numerous studies have examined the difference between EBM and ABM by using them to simulate systems in different domains [141, 149, 106].

In the commercial and business domain, Thawornchak [149] investigated the dynamics of supply networks by simulating this system using both agent-based and equation-based models. He designed a model of the same scenario using each approach and focusing on the main categories: producer, dealers and consumers. In the EBM approach, he used equations to identify the relationships between these three categories to generate the simulation of the real system. In the ABM simulation, he focused on the behaviour of every category as the beheviour of an agent and how this agent communicates with other agents in the system. From this experiment, he reached several findings: first, EBM can be more easily implemented due to software with diagram-based interfaces that makes specifying models simpler without requiring any programming skills. Second, simulation results are easy to understand in EBM because this approach uses both graphs and data to represent simulations. Third, ABM is more flexible in running models that follow predetermined actions and in presenting material and information flows. Finally, ABM allows modellers to study a system's behaviour in more detail, identifying any issues that may affect the system as a whole. This final observation is vitally important as it means that ABM is more practical for use in business to simulate some events that may occur in real life.

Table 2.1: Comparison of EBM and ABM

Criteria	ABM	EBM
Compute requirements	Intensive	Minimal
Interactions within models	Explicit	Implicit
Time and Events	Discrete	Continuous
Level of model complexity	High	Low
Practical use	Observing the emerging behaviour	Evaluation of what-if scenarios
Level of detail in implementation	High	Low

In social dynamics research, Tang et al. [147] have investigated the impact of using ABM rather than EBM to model human education data. They implemented ABM from existing equation-based models using Repast and then compared the results from both approaches. As a result of this study, Tang

et al. concluded that ABM could identify predictions that were not found by EBM. ABM could be verified by comparing the results with predictions made by EBM.

In epidemiological research, Sukumar and Nutaro [141] have used EBM and ABM to design a model to validate epidemiological disease-spread models, using the 1918 Spanish flu as a case study. EBM provides more intuitive interface than ABM for presenting facts. On the other hand, it was found that ABM is more versatile, allowing users to easily update the model by changing the network interactions, for example by adding a vaccination campaign to the model; subsequent changes in the epidemic can then be observed. Table 2.1 shows an overview of the comparison of EBMs and ABMs based on number of criteria reported by Sukumar and Nutaro [141].

To summarize the above literature review [141, 149, 106, 147, 44], the difference between EBM and ABM is as follows:

- **Flexibility:** The dynamic nature of the real world requires flexibility in presenting and simulating systems. It is necessary to make changes during the simulation process to represent updates that may occur in real life. ABM is more flexible in running a model, which can easily maximise the size of simulation and can also minimise it by adding or deleting agents during the simulation. Every agent in the ABM approach can decide how to communicate with other entities, which allows users to easily add or move any specification to the agent to produce new results.

- **Reality:** Individuals in the real world communicate simultaneously. Using the ABM approach to simulate real systems could help maintain this concurrency within the system because it can run the simulation on distributed architectures, making agents' communications occur concurrently.

- **System level or Individual level:** ABM is based on individual behaviour, and each agent has their own variables; this enables the analysis and validation of individual agents. In contrast, the EBM approach is based on the system level, which represents the interactions between a model's variables as a set of equations. These equations represent the system as a whole.

- **System Representation:** ABM represents the system as a series of agents whose behaviour creates communication with other agents, while EBM represents the system as equations that create the relationships between the system's variables.

2.2.5 ABMS platforms

Scheduling events and memory management are the most significant challenges facin interactive real-time applications like simulators. The main concept of scheduling is determining how a model's events are presented in time and how event execution order effects a simulation. However, depending on the scheduling choices, the computation of each event may or may not fit into the available memory, as "memory and storage have always been a limited parameter for large computations" according to [90]. Side by side, the cost of memory load during simulation time may also affect the overall performance. In many ABMS platforms, the schedule of events follows a predetermined order set by the modeller. This section presents the most popular platforms in ABMS, which include Swarm, NetLogo, Repast and MASON. As for and in more detail FLAME and FLAME GPU, they will be studied in more detail in the next chapter.

Swarm Swarm is the first and one of the most popular ABMS platforms[7]. The Swarm platform follows the 'framework and library' paradigm [117]. All of its libraries are written in Objective-C. It uses an object-oriented representation, which models agents as objects. Agents within Swarm interact with each other through discrete events to execute a schedule of actions. Swarm has a variety of methods for scheduling the order of these actions. The simplest scheduling method, called action group, is a collection of actions executed either in a linear sequence or, if suitable hardware is available, in parallel [96]. Memory space is managed within Swarm based on methods for allocating and freeing memory. Every object in Swarm is created in the memory space called the zone. The zone is responsible for tracking every object created there. To reuse memory, for instance, a signal can be sent to all objects for self-destruction [59].

NetLogo is a popular tool for the modelling and simulation of complex systems in the fields of natural and social sciences. NetLogo is written in Java,

and its modelling language is based on the Logo programming language, a member of the Lisp family[152]. Modellers can easily create their models using the model interface. Agents' behaviour and actions can be added through the procedure window. Programming in NetLogo is not object-oriented, and events are scheduled by default. The tasks are executed following the order of actions as they appear in the "go" procedure window [117]. For this reason, Meyer [93] classified NetLogo as a toolkit that tends to constrain model execution to the time-driven approach. NetLogo is designed to support learning and provides a simple way to create a simulation. The simulation runs sequentially on a single machine, but a BehaviorSpace tool is currently being integrated with NetLogoto support parallelisation. By default, a model runs in parallel, one per core, if the user has multiple processor cores. Memory is managed in NetLogo automatically as written in Java language. Java has a garbage collector that can delete unused objects and free up some memory automatically. However, not all objects are eligible for garbage collecting even when not in use. For this reason, there are two stages to follow to optimise memory usage in NetLogo. In the first stage, the memory needed for the model is estimated; then, the memory used is minimised. To calculate how much memory the model should consume, users must follow the formulas that appear in figure 2.3. After amount of memory needed is estimated, users can reduce memory use during the simulation by auditing the NetLogo code and removing unused agents and variables.

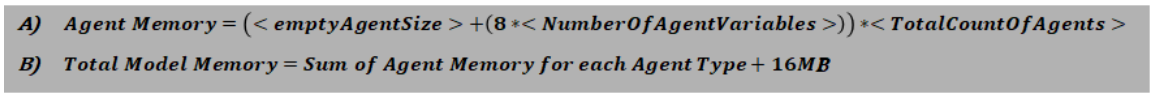

Figure 2.3: Formula A calculates agent memory. Formula B calculates model memory

The Recursive Porous Agent Simulation Toolkit (**Repast**) follows a scheduling schema adapted from Swarm. Repast is an open source ABMS platform started as Java version of Swarm, later becoming a separate modelling tool [117, 7, 29]. Repast **Simphony** and its new version, Repast HPC, are based on an OOP approach to specifying agents. The former is implemented in Java while the latter is implemented in C++. Events in both versions are driven by a discrete-event scheduler. Events are scheduled as a unit of time known as a tick. The scheduling mechanism used by Repast consists

of three classes, which are used to manage event execution. The main class is called `Basic Action` and contains all the basic actions set explicitly by the modeller or set implicitly via schedule objects. The second class is called `Schedule`. This class consists of objects and stores all the basic actions and information about these actions' execution time. The third class is called `Action Group` and is responsible for grouping basic actions. It provides methods of determining the order in which basic actions within the `Action Group` will be executed [27, 29]. The dynamic discrete-event scheduler used by Repast allows users to schedule events to be executed at a specific time, such as at the start or the end of the simulation [29]. For a large simulation that contains many agents or graphical components, the memory available to the Java runtime must be increased. By using the text editor, the arguments `-Xms` and `-Xmx` can be updated to allocate more memory in a similar situation. Repast version 2.6 and above JVM can automatically re-allocate memory up to the maximum system availability.

The multi-agent simulation of neighborhoods (**MASON**) is a free, open source multi-agent simulation toolkit. MASON is also based on the discrete-event simulation approach and is written in Java [82]. MASON was designed to serve a multi-agent simulation with a high number of agents. Also, this toolkit was implemented to work efficiently on single machine. MASON has a model library and separate 2D and 3D visualization tools, which allow the modeller to efficiently run up to a million agents [80]. MASON is written in three layers: the utility layer, the model layer and the visualization layer. The model layer and the visualization layer are independent, each having its own scheduler class. In the model layer, the `SimState` class contains the random number generator and the simulator's schedule to organise the order of the objects' occurrence at any real-valued time. In the visualization layer, the `GUIState` class contains a mini-schedule, which is kept in sync with the model's basic schedule. This allows two layers to work separately [81]. The large amount of memory required to run a complex simulation is one of the main limitations of MASON. D-MASON was created to overcome this problem and run simulations that are impractical or impossible to execute on a single computer [31]. D-MASON is the parallel version of the MASON library and allows modellers to run simulations on distributed hardware. D-MASON adds new layer called D-simulation,

which uses the usual visualization layer that exists in MASON and also leverages the specific functionalities of the D-visualization layer that supports data distribution[30]. As the simulation is distributed over several processors, each processor has its own distributed memory and the memory management technique required to minimise the time of memory access. Managing memory access is one of the most critical issues facing distributed systems.

2.2.6 Scalability of simulations in ABM

With the increasing complexity and increasing number of simulated entities, a simulation environment's scalability is a key indicator of its ability to cope with the complexity of the modelled system. In general, scalability can be defined as the property of a system compared to some other property of the same system [79]. According to [79] scalability is a major requirement when selecting a platform to deal with highly sophisticated and extensive models. Many large simulation experiments have been conducted with millions of agents [26, 28, 129, 68] and thousands of runs [6, 118, 61]. These studies focused on increasing both population size and the number of times the simulation was run. Some research has also focused on scaling the simulation execution time, such as [158].

Lorig et al. define workload and complexity as two key factors affecting the scalability of agent-based systems. The workload is derived from the amount of memory and the CPU threads used by the multi-agent framework. Complexity, by contrast, defines an agent-based system's computational effort [79]. The scalability of any modelling application can be analysed by observing how this application performs as the problem size increases. To measure system scalability, performance metrics have to be defined. This thesis's central measurement unit (in the implemented models) is the execution time of simulation experiments as the population size increases.

2.2.7 Agent-based models and parallelisation

The term 'large-scale' in ABMS refers to the high number of agents within the model, but it may also refer to the problems that could arise from model complexity or how a single processor can deal with a massive number of instructions during simulation [105]. To make the simulation more realistic and to represent more complex environments found in real life, modellers add more agents, ex-

tra rules and extra parameters to examine the whole system's behaviour. The increased complexity of ABMS has stimulated researchers in this field to look for solutions to accelerate the computation when a simulation is run. A number of methodologies have been developed to support ABMS platforms in managing complex problems. A number of ABMS platforms reviewed in the previous chapter support large-scale modelling in different ways, using parallel computing techniques to deal with the complexity of the modelled systems. Examples of such systems include FLAME[2], Repast HPC and D-MASON.

Parallel computing techniques require special architectures to allow tasks to be executed simultaneously. These include multiple instruction multiple data (MIMD) architectures and single instruction multiple data (SIMD) architectures. In MIMD, different instructions can be executed on different data on multiple independent processors simultaneously. Examples of this are found in clusters of computers and GRID computing. In contrast, SIMD architecture allows the same instructions to be replicated across multiple processors and on different data. MIMD architecture's high cost is its main drawback, which is why only large institutions can support it. In the 1990s, the use of MIMD architecture in clusters and GRIDs became more popular because they offered more powerful computation at low cost. Recently, the use of SIMD architecture returned with new processing architectures, especially general-purpose computing on graphics processing units [37].

GPU computing allows the CPU and GPU to operate together, producing HPC solutions for processing data more quickly and with a high throughput. More recently, a number of platforms and programs have been invented to provide an easier approach to GPU programming. Running ABMS applications on GPU is an affordable way to achieve high performance. FLAME GPU (section 2.3 reviews FLAME GPU in more detail) is an example of a framework that uses GPU architecture to run ABM applications. The development of ABM frameworks is ongoing, and researchers in this field are investigating new ways to apply them to HPC.

[2]http://flame.ac.uk/

2.3 Agent-Based Modelling on the GPU

A significant body of research focuses on the optimisation of agent-based models using GPUs. Most of this work tries to solve specific problems using GPUs; this process requires in-depth knowledge of GPU programming. Developing a dedicated platform for GPUs has the potential to solve a number of problems would otherwise have to be solved individually. This section presents a number of ABM models, techniques and platforms that can be run on GPUs.

2.3.1 GPU programming languages

Previously, GPU devices were accessed through graphics application programming interfaces (APIs) to render images and videos. More recently, researchers and developers have found a way to present their applications to the GPU using a set of fixed functions (low-level language) through DirectX and openGL. These APIs were originally used to design games and videos. However, the need for high levels of computation has encouraged the development of DirectX and openGL for programming platforms. Developers have also been motivated by the need to run their imaging applications in different fields through accessing GPU cards[103]. However, in reality, programming applications that use DirectX and OpenGL require specific knowledge to operate on the GPU, which is especially difficult for people unfamiliar with programming graphics cards [140]. Another demand of the GPU relates to sequential instructions. These are difficult to execute, which means there is a need for a new system through which to access GPU devices. For this reason, NVIDIA and AMD developed new systems (CUDA and OpenCL) that are easy to use. These systems provide direct access to the GPU, making it more programmable than before [103].

OpenCL is a GPU programming model and a type of API platform that allows parallel programming for heterogeneous systems, such as CPUs, GPUs and other processors from different vendors [65]. OpenCL was initially proposed by Apple. In 2008, it became available to the public after being tested by the Khronos Group. The major feature of OpenCL its threading model similar to the one that exists in CUDA, consisting of a similar hierarchical memory structure [17]. OpenCL enables two types of parallel processing: data-based and task-based[140].

It is worth clarifying that CUDA, another GPU programming model, was

designed only for NVIDIA's graphics cards. CUDA has a massive parallel structure, hierarchical memory architecture and threading programming model. All these features make GPU cards accessible and allow the CPU and GPU to work together to process one program.

CUDA performed more efficiently than OpenCL in terms of the transformation of data between the CPU and GPU. Karimi et al. [65] and Su et al. [140] conducted similar investigations whose findings supported this efficiency. They compared CUDA and OpenCL through the use of specific real-world applications. The results of the Karimi et al. study revealed that the execution time of CUDA's kernel is faster than OpenCL's although the two implementations run nearly identical code. Su et al.'s [140] study compared the efficacy of C, CUDA and OpenCL by applying five application benchmarks. The results showed that the execution time of the CUDA driver API was 3.8-5.4 times faster than the execution time of OpenCL.

Running different applications using CUDA and OpenCL showed that CUDA performs 30 percent better than OpenCL at the most. To avoid generalizability, Fang et al. [43] clarified that OpenCL performs similarly to CUDA under fair-comparison conditions. This comparison requires more code for memory allocations and a different code for transfers of data for each type of hardware used in OpenCL. This exposes the fact that OpenCL must be enabled for any GPU device and requires a different optimisation for each hardware structure, while CUDA can use only NVIDIA GPUs that have the same hardware architecture as each other. Consequently, a similar optimisation code can be used with different applications. To summarise the previous literature, there is a clear preference for CUDA in relation to GPU programming in the current climate.

2.3.2 CUDA

CUDA is a programming model used for parallel processing. This technology was invented in 2007 to develop software for NVIDIA's graphics cards. The highly parallel architecture offered by CUDA allows instructions to be run simultaneously on hundreds of GPU processor cores [60]. This section will examine CUDA from the perspective of CUDA architecture, CUDA memory models and CUDA programming models.

2.3.2.1 CUDA architecture:

The CUDA architecture is represented by three basic components: threads, blocks and the grid. The grid consists of a number of blocks, and every block contains a number of threads. Blocks of threads within a grid can represent a one-dimensional, two-dimensional or three-dimensional thread. Threads within the block can represent one-dimensional or two-dimensional threads [60]. The number of threads per block and the number of blocks within a grid depend on the GPU architecture that CUDA accesses. The Fermi architecture is the most recent computing architecture for GPUs.

It consists of 512 CUDA cores (threads per block). The 512 cores are divided into 16 streaming multiprocessors (SMs); thus, there are 32 CUDA cores for each SM. Each CUDA core consists of an integer arithmetic logic unit and a floating point unit[60]. The maximum number of threads per block is 512 within the graphics card hardware support with a 1.x compute capability, and the grid can represent a one- or two-dimensional thread. In a graphics card with a 2.x compute capability, the maximum number of threads per block is 1024, and the grid can represent a one-, two- or three-dimensional thread [102].

Understanding the CUDA architecture allows programmers to write effective CUDA code. A software program in CUDA can easily access the hardware structure through four built-in variables representing the basic components of the CUDA device architecture. The number of `blocks` within a `gird` can be determined by the `gridDim` variable. The `blockDim` variable can present the number of threads within a block. The `blockIdx` variable is used to access the blockID, and the `threadID` can be accessed using the `threadIdx` variable. Threads within the same grid are run by the same kernel function, but they are not synchronised, whereas they are synchronised in the same block [102].

2.3.2.2 CUDA memories

CUDA consists of two kinds of memory structures, depending on their accessibility by the CPU and GPU code. The device (GPU) code can access the following memories:

- Register memory: used per thread for reading and writing data.
- Local memory: used per thread for reading and writing data.

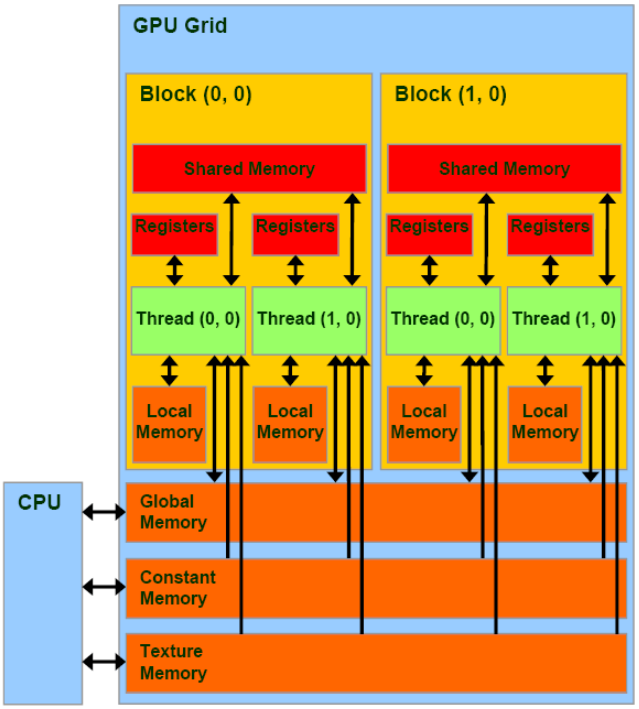

Figure 2.4: CUDA Memories[3]

- Shared memory: used per block for reading and writing data.
- Global memory: used per grid for reading and writing data.
- Constant memory: used per grid for reading only.
- Texture memory: used for reading only.

The host (CPU) code can access global, constant and texture memory to transfer data between the host and the device per grid. All these types of memory can be accessed by the host for reading and writing (see figure 2.4).

In summary, the CUDA memory architecture consists of registers, shared memory and constant memory, which can be easily accessed in parallel and with higher speed than the global memory. Making the best use of these types of memory and dealing with the limitations of the hardware will help in the design of efficient code [69].

2.3.2.3 CUDA programming model

The structure of the CUDA program consists of a serial program code that calls parallel functions (kernels). The CUDA programming model has the ability to execute the serial code on CPUs using normal instructions from the C/C++ program and to execute the parallel code on GPU devices by calling up the

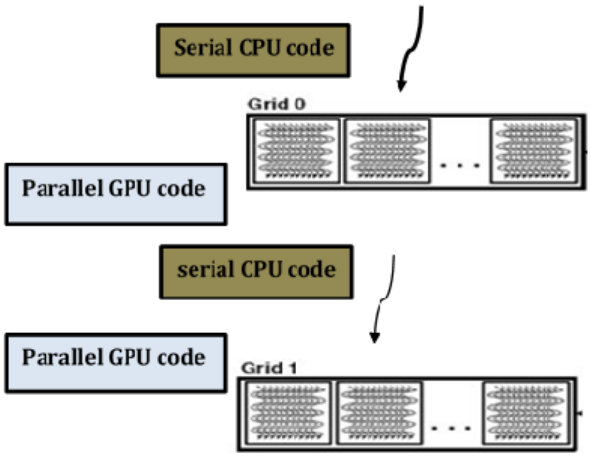

Figure 2.5: CUDA execution program

kernel functions [46]. Figure 2.5 shows the CUDA execution program.

In the CUDA programming model, the CPU is called the host and the GPU card is called the device. The CUDA program starts in the host as a normal C/C++ program and contains the usual main functions. Launching the kernel function requires a specific function specially designed to enable the GPU card. For example, *-global-* is a CUDA keyword that refers to all functions called up from the host to be executed on the device. *-device-* is another CUDA keyword that refers to all functions that are called up and that run only on the device. A specific function is also required to access the allocated and free memories on the device and to transfer variables between the host and the device. *cudaMalloc()* is used to allocate memory, *cudaMemcpy()* is used to copy data between the host and the device and *cudaFree()* is used to free up allocated memories. The kernel is launched from the main() function in the host and requires a triple-angle bracket to be written and filled with three configuration parameters representing the number of blocks per grid, the number of threads per block and the total required shared memory, which is an optional parameter [69, 46]. A number of steps must be completed each time for any code is run on the device. The steps listed below must written in the host:

- Declare all variables needed in this kernel for the CPU and GPU.
- Allocate a memory device for these variables.
- Copy these variables from the host to the device.
- Launch the kernel function, as mentioned previously.

[5]The source for figure 2.4 is http://cuda-programming.blogspot.co.uk/2013/01/what-is-constant-memory-in-cuda.html

- Copy the result back to the host if there is a result.
- Free up the used memory on both sides.

The CUDA program will be compiled on an NVIDIA C Compiler (NVCC). The NVCC will divide the program code into two parts. The CPU code will be compiled using a standard C compiler, and the GPU code will be compiled by the NVCC [69].

Exploiting the processing power of GPUs for the simulation of complex systems offers a high computational power. Based on the reviewed literature on CUDA technology, running a simulation application using CUDA seems to be more a complicated process for modellers who are not familiar with GPU computing. Flame GPU allows modellers to run their simulation models on GPU without explicit understanding of CUDA programming.

2.3.3 Efficient performance of agent-based simulation on GPU

In the field of ABM, creating a simulation by modelling individuals helps build a natural and flexible environment in which to study a system's behaviour, but this process requires more computational power. Traditionally, ABM platforms use serialised algorithms in their structures to run simulations and manipulate mobile discrete agents. However, this technique limits simulation speed and model scalability [124]. This implies the need for an HPC environment or specialised workstation of parallel or distributed platforms [2]. Much research has focused on enhancing the performance of ABM platforms using various strategies. For instance, simulations can be distributed to minimise simulation time; the distributed simulations of multi-agent systems can be implemented using a dedicated computing cluster [144, 143, 97, 150] or a grid [151, 110]. However, the increase in performance achieved by applying CPU parallelism using distribution techniques may be affected by a number of issues, including the management of communication between dynamic resource allocations and nodes and the monitoring of the state of the distributed simulation. The ability of the shared memory parallel architecture of a GPU to run simulations could be exploited to overcome many of these problems. A number of studies have focused on using GPUs to implement ABM simulation, such as [75, 129, 83].

The next section presents examples of GPU agent-based models, frameworks and techniques.

2.3.3.1 ABM models on GPU

To enhance their performance, a number of ABM models in different domains have been re-implemented for GPU. A significant improvement in execution time has been found when models optimised for parallel GPU execution are used, compared to the use of a sequential model. In the field of biology and medicine, Chen et al. [21] present the simulation of blood coagulation using GPU, and, for comparison, the simulations were implemented in NetLogo, Repast and a direct C version. The experiment results showed that the version with GPU implementation is 10 times faster than the C version, over 100 times faster than the Repast version and over 300 times faster than the NetLogo simulation. Alberts et al. [4] implemented a version of the Toy Infection Model (systemic inflammatory response syndrome) using CUDA and compared it with the original version implemented in NetLogo. The parallel version of the model offers a substantial gain in performance compared to the original version with no loss in accuracy. The use of GPU allowed Campeotto et al. [19] to increase the speed of simulating a protein structure prediction problem by up to 36x. They used concurrent agents to explore the folding of different parts of a protein. Less time is needed to find a solution in the new version (minutes) compared with the CPU version (hours).

In the field of physics, the molecular dynamics (MD) simulation that was implemented using parallel algorithms on the hybrid CPU-GPU platform performed better than the previous parallel algorithms on the CPU cluster platform [77].

Ant Colony Optimization (ACO) is acknowledged as a powerful method of finding the best solution to many optimisation problems, such as graph theory. Several studies have presented a GPU implementation of the ACO algorithms [20, 35, 33]. Each one of these studies focused on a specific problem and used a specific technique to implement the GPU. The GPU implementation of ACO offered better results in both speed and solution quality with some limitations of GPU memory for large simulations [35].

2.3.4 Techniques to implement ABMs on GPU

The latency-hiding mechanism is one such technique used to implement ABM simulation on GPU. Aaby et al. [1] approved the efficiency of this mechanism. They separated the grid of agents into dependent blocks allotted to independent processing elements. After updating agent states in each block, the data dependencies between these blocks will be exchanged. This process takes advantage of the fast shared memory that can be accessed by threads in the same block. However, the large size and greater complexity and dimensionality of some simulations required intensive coding to apply this technique to GPU. Hermellin et al. [56], [57] follow another technique based on a hybrid CPU/GPU approach, which means the simulation's execution can be divided between CPU and GPU. In this approach, a clear separation is made between the agent behaviours (handled by the CPU) and environmental dynamics (managed by the GPU). Leaving part of the simulation on the CPU increases memory transfers between the CPU and GPU and decreases parallelism. This paper [108] presents three approaches using GPU to implement multi-agent simulation. All approaches are based on the concepts of task allocation using the GPU. The authors mentioned possible bottlenecks in each approach, but the main challenge is data transfer overheads between CPU and GPU.

2.3.5 ABM frameworks using GPU

A few ABM frameworks use GPU: the TurtleKit framework [94], the many-core multi-agent system (MCMAS) framework [74], the MASS CUDA library[72, 54] and the FLAME GPU framework.

The TurtleKit framework is the future third version of the TurtleKit simulation platform[95] that enables GPUs to handle environment dynamics while the simulation of agent behaviours are implemented on the CPU. Agent behaviours that do not depend on the agent state will be performed on the GPU to increase performance and reduce the impact of the GPU facilities on the ease of maintaining the simulation code. However, the parallelism of this platform is limited by the use of a hybrid approach and the division of the simulation between the CPU and the GPU.

The MCMAS framework is based on two approaches. The first is the use of a high-level Java interface to OpenCL code and the commonly used func-

tions and data structures (plugins) that are provided by the framework. The second approach is the ability to define new plugins for MCMAS as OpenCL code. The MCMAS library provides a set of famous algorithms, such as diffusion, path-finding and population dynamics, that can be easily used. However, the limited number of predefined data structures and functions supported by MCMAS require low-level GPU coding to extend some modules.

MASS is a parallelising library for multi-agent spatial simulation [45, 24] developed by researchers at The University of Washington[4]. This project provides Java, C++ and CUDA implementations of the MASS library. Currently, MASS CUDA library developers are focusing on analysing typical agent behaviours and developing GPU parallelisation techniques to address these behaviours.

2.4 FLAME GPU framework

FLAME GPU [128] is an extension of the FLAME[5]framework using GPU computation. FLAME GPU follows a formal ABM specification schema called an X-machine [32] that allows modellers to design a wide range of agent and non-agent models. An agent-based model in FLAME GPU consists of collections of agents of different types, interacting within an environment where each type of agent is specified abstractly as an extended finite state machine. The particular EFSM model chosen is the X-Machine. Modellers define the agents, their functions and internal memory in the X-machine Mark-up Modelling Language (XMML) and use the FLAME GPU templates system to generate a simulation program that can efficiently run the code in parallel on the GPU.

According to Tamrakar [145], there are three advantages of using the FLAME GPU framework to design a model: first, modellers can implement rules for agent behaviours and simulate ABMs on parallel architecture (GPU) with a minimum understanding of parallel computing or any experience of programming GPUs. Second, the computational performance achieved when simulating massive agent populations on GPU is far greater than when executed on CPU. Finally, locating agent state variables in GPU memory makes visualisation more efficient, as it is not necessary to copy the data from RAM to the GPU for rendering.

[3]https://depts.washington.edu/dslab/MASS/

[4]The Flexible Large-scale Agent-based Modelling Environment (FLAME) is a template-based simulation environment designed by Coakley et al. [25, 23].

2.4.1 Code generation in FLAME GPU

The FLAME GPU framework consists of a number of X-agents (the agent representation of an X-machine [29]) specifications. Each instance of an X-agent has its own memory that holds a set of variables. All instances of X-agents have a start state, an end state and transition functions that can read and write to their memory. Agents can communicate by sending and receiving messages, and their functions can read and write these messages at any time between the start and end states for each agent. The process of creating a model using FLAME GPU is very similar to that of creating the original FLAME, which required the model specification to be written in XML format within an XMML document. However, the syntax used to write the model in FLAME GPU uses an extended version of the FLAME XML schema. The GPUXMML extension outlines the GPU-specific model description elements, such as the maximum size of an agent's memory [9]. This allows a formal agent specification to be transformed to optimised C-based CUDA code through GPU-specific templates. CUDA technology is a framework that allows programmers to develop general purpose algorithms for the GPU. FLAME GPU uses extensible stylesheet language translations (XSLT) to parse XML files, thus avoiding the problems caused by XParser program in FLAME [125]. XSLT is a flexible language used to translate XML documents to other formats, such as HTML or other document formats. XSLT templates can be processed by any compliant processor, such as Saxon, Visual Studio, Xalan or xsltproc. Figure 2.6 shows the processing steps to generate a FLAME GPU simulation program.

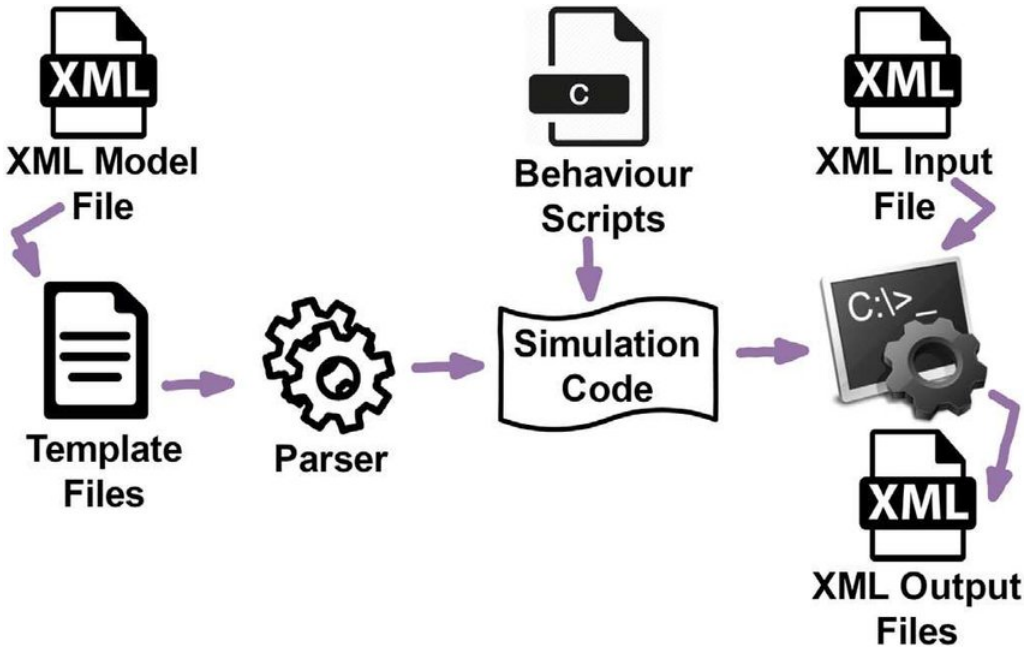

Figure 2.6: The FLAME GPU-modelling and code-generation process. Figure after [22].

2.5 Impact of Data dependencies in Real-Time High Performance Computing

Data dependency analysis is a method of detecting parallelism between two blocks or two statements. It is the key to optimising the code and provides dependency information that can help to improve memory locality, load balancing and efficient scheduling. In the area of high-performance computing, data dependency analysis aims to improve performance by improving both interprocess communication and accessing shared memory location [113, 114, 107]. Dependency analysis is a promising approach to detecting problems in large systems. For this reason, starting in 1979, several authors have proposed methods and techniques focusing on this matter in both data dependency and task dependency [55, 109]. A number of studies also compared the accuracy and efficiency of these techniques.

According to Patil and Jagtap[107], data dependency appears when two statements following each other share the same memory location and one of them writes to it. The authors classified dependency into three classes: true dependency, anti dependency and output dependency. In true dependency, statement 1 writes to the memory location while statement 2 reads from it (read-after-write). In anti dependency, statement 1 reads from the memory lo-

cation while statement 2 writes to that location (write-after-read). In output dependency, both statements write to the same memory location (write-after-write). All these types of memory access are performed with scalar variables, array variables or pointer references in a sequential program. These dependency problems can be solved using a data dependency testing technique, such as the Banerjee Test[154], I-test[71], GCD Test[153] or Omega Test[115]. These tests are examples of code-based dependency analysis methods, according to Jagtapa and Shrawankar[63]. They classified dependency analysis approaches into three groups based on the input source: a code-base approach, a component-based approach and a run-time analysis approach. The next section details these approaches and gives some examples covering various application areas in different computing environments.

2.5.1 Data dependency techniques

Different dependency analysis techniques have been proposed in various areas. Some of these methods are used to identify dependency relationships between statements in a program, and some are used to work on the relationships between nodes in a large system that has heterogeneous hardware and software components. This section follows [63]'s method of classifying dependency analysis techniques, which is based on the type of input source of information.

2.5.1.1 Code-based approach

The available methods that followed code-based approach can be applied to program or code instructions. This approach helps to identify dependency between statements, loops and variables in a program and can be used to parallelise compilers and sequential code and for natural language processing systems. Various methods available in this type, such as [154, 71, 153, 115, 160, 78, 15, 159]. Below is a brief description of a number of well-known data dependency tests.

GCD Test: The greatest common divisor test is used to test data dependency between loop statements: "It is based on a theorem of elementary number theory, which states that a linear equation has an integer solution if and only if the greatest common divisor of the coefficients on the left-hand side

(LHS) of the equation evenly divides the constant term on the right hand side" [113]. If there are no integer solutions to the linear equation, then there is no dependency between statements. This test is unable to prove dependencies; it can only disprove them[107].

Banerjee Test: The Banerjee test is the most widely used test in compilers. It uses a simple concept and is efficient in disproving dependences. It is based on the intermediate value theorem, which computes minimum and maximum values of the equation on the left-hand side. If the constant from the right-hand side lies within the two extremities of the left-hand side, then the dependency exists. The direction information for the dependent instances is provided using this test.

I-test: The I-test is based on and enhances both the GCD test and Banerjee test but, unlike them, it disproves dependency. The I-test can positively prove or disprove the existence of an integer solution in most cases. This test was designed to analyse the dependency of linear subscripts. It is a polynomial-time test, which can detect integer solutions for single-array subscripts with constant bounds [73, 107].

Omega Test: The Omega test is an integer programming algorithm used to detect the dependency between two array references. It is used to determine whether there is an integer solution to a dependence equation. This test can also handle many data dependence problems, such as symbolic variables, nested if-statements and triangular, trapezoidal bounds [107]. The Omega test is based on an extension of both the least remainder algorithm and Fourier-Motzkin variable elimination [48]. In terms of accuracy, the Omega test can prove and disprove more dependence problems than the Banerjee test and the I-Test[114], but it has a higher computation cost in order to be exact[73].

2.5.1.2 Component-based approach

In this approach, system representation is based on models that describe the structure and behaviour of systems such as UML, ADL[6]or IDL[7]models. These models are used as input for different kinds of graph methods to represent the communication between system components, such as the relationship between

program functions. This type of approach is used to parallelise the systems and automatically partition software. Directed acyclic graph (DAG) and data flow graphs (DFG) are examples of this approach.

Directed Acyclic Graph: DAG is a type of graph scheduling used to solve a variety of computing problems. The application of DAG in scheduling will represent the tasks as DAG nodes and the data dependencies between tasks as DAG edges.

2.5.1.3 Run-time analysis approach

Both the code-based and component-based approach are used in the run-time analysis approach to detect dependencies in applications that require run-time monitoring. This section presents a number of systems used to detect dependency and then schedule tasks for the runtime of hybrid applications.

DAGuE: The directed acyclic graph unified environment is a system that allows scientific computing to work on distributed environments and heterogeneous systems, such as systems with many cores, accelerators and high-speed networks. It consists of a number of libraries, a run-time system and a number of tools to help an application's developers run difficult tasks in diverse environments. DAGuE uses job data flow JDF for the internal representation of DAG. This system has been merged with the PaRSEC (parallel runtime scheduling and execution controller) project [40].

PaRSEC: The main aim of the ParSEC system is to create a new path for scheduling tasks at run-time, which can be done using message-passing interface (MPI) programing. Within each node, the data generated from a completed task will be used to enable the execution of the next task in the right place. The decision that is made depends on the message relayed back by the completed task, which contains information about the available hardware resources. MPI programing divides the data on different processors into tasks, and these tasks can communicate by passing messages to each other.

StarPU: StarPU is a unified platform for scheduling and executing parallel tasks over heterogeneous hardware. It provides a unified runtime layer

for heterogeneous multicore processors and accelerator technologies. StarPU also provides a high-level library that efficiently transfers data between heterogeneous machines using MSI caching protocol. This technique will minimize data transfer and eviction heuristics to solve the limited memory availability issue. StarPU also offers a uniform approach to the parallelisation on heterogeneous architectures by the defined abstraction of tasks that can be executed asynchronously. Finally, the StarPU model works well on top of multicore processors, the Cell processor and CUDA-enabled GPUs [34, 44].

2.5.2 Reduce memory movement

In large parallel architecture systems, interconnections become more hierarchical; this hierarchy increases the memory access gap, affecting both system latency and bandwidth [111]. Reducing data movement (data transfer between processors and system memory) in such systems would improve their overall performance. A number of techniques and associated studies have focused on this goal; these include load balancing [138, 66, 67], graph partitioning [13] [146] and spatial partitioning (or spatial messaging) [129] [127]. Generally, the graph-partitioning algorithm evenly divides work among computation nodes to minimise data movement. To improve performance and reduce data transfer across the system, Barrera et al. [13] used the graph-partitioning technique. They automatically applied task-dependency graphs during system runtime to collect information and then used advanced graph partitioning to break the graphs into smaller parts. These partitions were used to minimise data movement across the shared memory system. To minimise data movement between processors and reduce workflow execution time, Tanaka and Tatebe [146] applied the multi-constraint graph-partitioning method to the workflow-directed acyclic graph (DAG), which represents task dependency. The graph-partitioning method in this study helped to decrease workflow execution time by 31% and reduce the remote file access from 88% to 14% of total file access.

To reduce memory transfer in large-scale MAS, a number of data structure accelerating algorithms have been used, one of which is spatial partitioning

[6] Architectural description language

[7] Interface description language

[76]. The main aim of the spatial-partitioning technique is to reduce the communication overhead of the simulation agents. It needs only a subset of interaction to allow reducing memory movement. This technique has notably been used in interacting systems, such as swarm-based systems on GPUs [127] [41], on computing clusters [50] and on the PS3 [119]. The spatial-partitioning algorithm was used to minimise the number of messages read by each agent based on the interaction radius of the message or particle [129].

Load balancing is another strategy to reduce data movement or balance a compute load, especially in distributed applications. According to Mishra [98], load balancing helps avoid too overloading the resources and minimises the total waiting time for the resources. A number of studies have discussed the concept of load balancing and improving system performance and efficiency, such as [138, 66, 67]. In [138], an enhanced dynamic load-balancing algorithm is proposed to improve performance in grid computing, whereas [66] and [67] reviewed the implementation of a number of load-balancing algorithms in cloud computing and discussed their effects on cloud-computing applications.

2.6 Summary

This chapter presented a review of ABM and simulation in general and on the GPU, with some examples of both types. This was followed by an overview of the FLAME GPU framework and the way its code is generated. The impact of data dependencies in real-time high-performance computing were also reviewed, as were the existing dependency analysis techniques.

All the platforms reviewed above (except NetLogo) follow an OOP approach to building models. The dynamic relationships between agents in simulation require a flexible technique to create a suitable environment in which agent behaviour can occur. In contrast, OOP has a fixed technique when objects communicate with each other following predefined relationships within the class. A number of these platforms do not have the ability to run single models across multiple computers without extra support, such as BehaviorSpace tool in NetLogo or the new functional layers in D-MASON to support data distribution in MASON. FLAME, on the other hand, is based on the formal method of model specification called X-machines, which provides a technique for both specifica-

tion and validation [7, 129]. This technique allows developers to create models and software tools that are compatible with each other, and that, with little effort, can be used to develop new models based existing models[7]. In some of these platforms (NetLogo, MASON) and similarly they lack support for execution on the GPU. GPU is tailored to 3D graphics computations and requires specific programming platforms to run a general-purpose application, such as ABM, on it.

ABMS applications are inherently complex such that running a model's tasks simultaneously in parallel architectures, such as GPUs, provides a real environment for the simulation. While some research into the use of the GPU for ABM has been done, it has been limited to either fixing a specific problem or implementing a specific model. The use of the hybrid CPU/GPU model approach in both Turtlekit and MCMAS has led to decreased parallelism. This hybrid approach also requires frequent memory transfers between CPU and GPU. The MASS CUDA library is limited to improving the performance of spatial simulations and implementing popular agents behaviours. MASS CUDA still needs more work to be able to analyse typical agent behaviours and develop GPU parallelisation techniques.

The FLAME GPU framework offers the ABMS community the power of GPU computing without the need for specific skills. FLAME GPU users can easily write model specifications in XML and in C, and these will be compiled automatically to CUDA code. Furthermore, FLAME GPU accelerates the execution time of ABMS applications by up to 250X faster than CPU execution time [37]. The complexity and scalability of a large simulation requires more processing power to increase the number of calculations per unit of time and memory available to allow a high number of agents. Memory restrictions were one of the motivations for researchers to develop platforms that support parallel and distributed environments, such as Repast HPC and D-MASON. However, even with the extra memory offered by the new environment, managing memory access is one of the most critical issues facing distributed systems.

Data-dependency analysis is a method of detecting parallelism between two blocks or two statements. Prior knowledge of data dependency is key to optimising the code and providing information that minimises memory access. In the reviewed literature, various approaches have been applied to detect dependency, including the component-based approach and run-time analysis approach. Such

approaches could be used to explore the dependencies within the FLAME GPU models and thus reduce memory movement.

Chapter 3

Agent-Based Models for the GPU

The previous chapter reviewed a variety of topics that cover the background of this research. These topics include all related work on ABM and simulation, ABM on GPU, an overview of FLAME GPU and the impact of data dependencies in real-time HPC. This chapter focuses on the FLAME GPU framework in more detail, and includes the design and implementation of models using Flame GPU. The data handling mechanism and inter-agent communication within this platform will also be discussed. Furthermore, this chapter also shows how to implement the Boids model using FLAME GPU.

3.1 Designing X-Agents Using FLAME GPU

3.1.1 X-machine

The X-machine was first proposed in 1974 by Samuel Eilenberg [39] as one of several different extensions to a basic Finite State Machine. The innovation in the X-machine is that its transitions are labelled by processing functions that act on memory. The X in (X-machine) represents the basic underlying data type processed in a model. For example, when an X-machine is used to describe a biological cell, X will represent the memory of the cell unit, such as a vector of attributes; however, in general, X may represent any data structure. X-machines are extended finite state machines with memory. A FSM is formally defined as the 5-tuple (Q, Σ, q0, F, T), where:

- Q is a finite set of states

- Σ is a finite alphabet of input symbols
- q0 is the initial state where q0 is a member of Q
- F is the state transition function from Q $\times$ Σ to Q
- T is a set of final states where $T \subset$ Q.

X-machines are extended FSMs (EFSMs) in that they also have memory and a tuple of variables, which may be read or updated when executing a transition. The particular variant of X-machine adopted here is the Stream X-machine (SXM), which also processes inputs and outputs when executing a transition. In the biological cell example, X-machine memory will hold all cell information, such as cell cycle, cell size, cell position and cell bonds. The transitions between states also update the cell's variables in memory, for example, to move the cell, 'cell position' will be updated, and to grow the cell, 'cell size' will be updated [25]. The state transition function, besides updating memory values, will receive input symbols from the input stream and produce output symbols, which will be part of the output stream. The formal definition of a stream X-Machine [32] is a 9-tuple = (Σ; Γ; Q; M; Φ; F; I ; T; m0), where:

- Σ and Γ are the input and output alphabets.
- Q is the limited set of states.
- M is a infinite set called memory.
- Φ is a set of partial functions φ; each function of this type maps an input and a memory value to an output and a possibly different memory value, φ: $\Sigma \times$ M $\rightarrow \Gamma \times$ M.
- F is the next state partial function, F : Q $\times$ $\Phi \rightarrow$ Q. F is often described as a state transition diagram.
- I and T are the sets of initial and final states.
- m0 is the initial memory.

Adding the ability to X-machines to communicate with each other can be achieved by using communicating X-machines (CXM). The general definition of a CXM model that is able to exchange messages is the tuple [25]: $((C_i^x)i = 1..n, R)$, where:

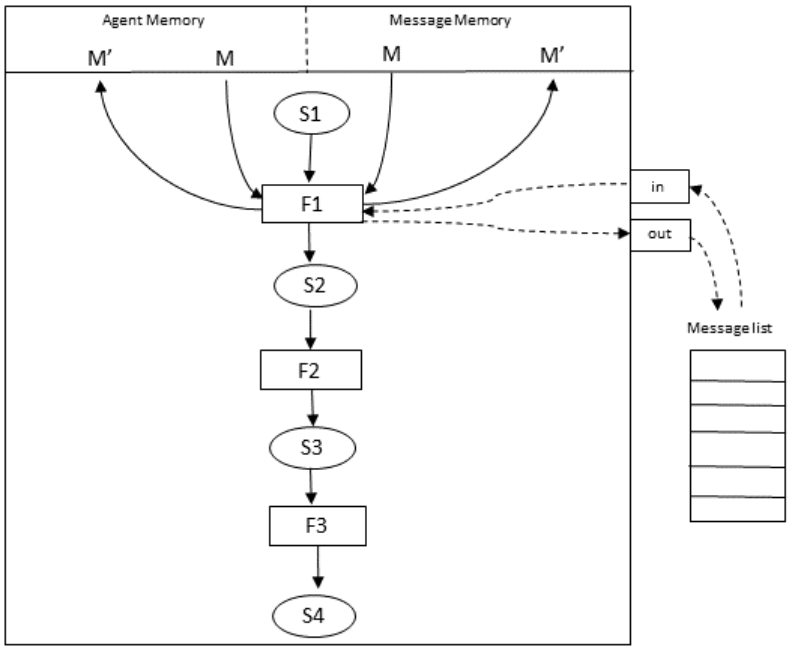

Figure 3.1: Stream X-machine specification, M and M represent the agent memory set before and after agent function F1, which inputs and outputs messages to the message list. [126].

- C_i^x is the $i-th$ communicating X-machine in the system, and

- R is a communicating relation between the n X-machines.

In FLAME GPU, an agent is represented as a form of state machine that consists of internal memory (M as in the formal definition), an agent's functions (next state partial functions, F, in the formal definition) and a set of states (Q as in the formal definition); X-machine agents can only communicate through messages. This can be observed in Fig 3.1. This state diagram represents a set of states, functions and input and output data that can be processed through these functions. The agent's memory can be updated every time step that is needed through this process. The smallest unit that can be processed is an agent, and whenever agents communicate with each other, all agent memory needs to be copied in every transition function from one state to another.

3.2 FLAME GPU features:

3.2.1 Agent Data Storage and Access

As mentioned above, the formal representation of an agent within the current FLAME GPU is based on the concept of a communicating X-machine as with the original FLAME. Three major components are needed to execute a model using FLAME GPU: agents, messages and layers. Agents present the agent description, messages show how the agents communicate with each other and layers show the order of agent behaviour during the simulation. Both agents and messages have their own memory that holds agent properties and information that needs to be passed between agents. Within FLAME GPU, every GPU thread represents a single agent, and a (GPU) device wrapper function is used for each agent function to hide GPU memory access [126]. However, even with special techniques for hiding the cost of memory access, GPU memory bandwidth is a limited resource in large and complex models.

Within FLAME GPU, each agent function (the main representation of agent behaviour) is represented by a unique GPU kernel. Using this process, global synchronisation of the entire agent population is ensured after each transitional stage. Agent data in parallel threads are stored temporarily in both the fast multiprocessor register, and shared memory. When moving data from global device memory, an array of structures (AoS) is used to allow more efficient memory access for both reading and writing data. GPU memory coalescing allows for more efficient use of memory; data is consecutively accessed and fewer memory requests are issued [126].

3.2.2 Birth and Death

The addition of new agents in FLAME GPU requires pre-allocated memory space. The entire agent population must be double buffered to provide sufficient storage space for any new agents. The process of agent births can then be achieved using linear mapping, where each agent in the current agent list outputs to the same global position. Within the sparse new agent list, a simple flag variable is used to indicate the presence of a new agent. This flag is then used to perform the inclusive prefix sum[53]. The total number of new agents

can easily be determined by considering the position of the last new agent.

The FLAME GPU uses stream compaction which uses a parallel primitive algorithm from the Thrust [58] to avoid sparse data. From the source array, stream compaction produces a smaller array and includes only the wanted elements. In parallel and without conflicted memory access writes, this compacted data stream can be used to write agents to newly compressed, unique locations.

3.2.3 Agent Communication

In FLAME GPU, agent communication can be achieved through the use of message lists. To ensure memory coalescing for agent memory, FLAME GPU utilises structured, efficient access to data. An abstraction of messaging is used for internal communication between agents. Message processing within FLAME GPU supports three different techniques for reducing the transmission of data which we refer to as the brute force distributed and discrete messaging techniques. Each agent reads every available message in brute force messaging; to accelerate this process, shared memory is used to load messages that are accessed by agents within a group of threads. Shared memory is much faster than global memory because it is located per thread block, thus allowing a group of threads to access the same shared memory [124]. In spatially-distributed messaging, agents can read messages within a fixed radius in a 2D or 3D continuous space. For this messaging type, FLAME GPU uses a parallel sort algorithm to reorder agents and build a matrix containing the start and end index positions of any agents within a fixed (message) radius that is sized to represent the agent's environment. After iterating through message lists within neighbouring partitions, only agents within the defined radius will be returned. Using texture memory[1]to load messages accelerates message reading, as shared memory cannot be used in this technique because agents are in different locations and access different messages stored in different positions. This is equivalent to previous work on data structures for reducing memory transfer. Within the discrete messaging technique, shared memory or the texture cache can be used to load a 2D discrete grid of messages [124, 129].

[1]More detail about texture memory is available through this link: http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

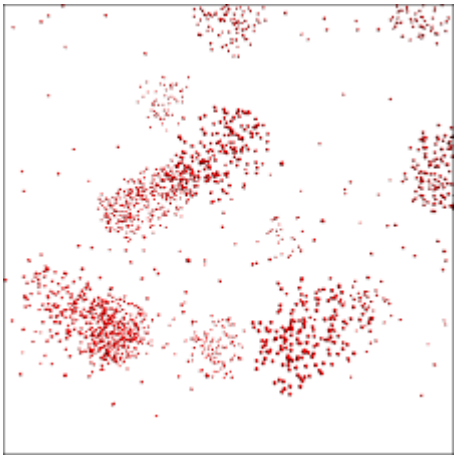

Figure 3.2: The visual representation of the Boids model.

3.3 Implementing a model using FLAME GPU

The following worked example presents the programming syntax required to implement a simple Boids model in FLAME GPU. The Boids model has been chosen for its simplicity. A number of examples of complex models are provided within the FLAME GPU SDK.

3.3.1 The Boids model

Boids is an artificial life model developed by Craig Reynolds [121, 119] that describes the behaviour of flocking fish or birds. According to Reynolds (2001), flocking is an example of emergence, by which the interactions of simple local rules produce a complex global behaviour. A visual representation of the Boids model is shown in Fig.3.2, which shows group flocking behaviour. There are three simple steering behaviours that an agent in the Boids model can follow: 1) alignment, which is steering towards the average heading of nearby neighbours; 2) separation, steering to avoid crowding nearest neighbours; and 3) cohesion, steering to move toward the average position of the immediate flockmates [120]. Fig.3.3 shows the state diagram for the implementation of the Boids model by the FLAME GPU that includes the agent functions (Input data , Output data, Move) that define the beahaviour of the Boid agents, the order of their execution and the location message that allows the agents to indirectly communicate information[121].

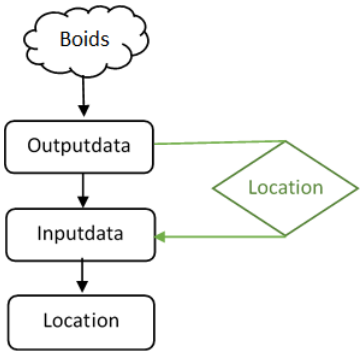

Figure 3.3: The Boids model state diagram showing the function dependency relationship of functions required to implement the required flocking behaviour.

3.3.2 Model specification

The implementation of the Boids model using FLAME GPU required two user-specified files. The first is an XML file called 'XMLModelFile.xml' which contains the model description. The second file, 'Function.c', is the agent behaviour functions file. There are three main components in the model description file: agents, messages and layers. Agents refers to the definition of what an agent is and consists of memory, functions and states. Messages refers to the information that the agent needs to communicate to other agents. Layers refers to the mechanisms that determine the order in which actions are performed. Fig 3.4 shows the outline structure of the XML code of the Boids model.

```
1    <name>Boids</name>
2  ⊞ <gpu:environment>
7  ⊟ <xagents>
8  ⊟   <gpu:xagent>
9        <name>Boid</name>
10       <description>a simple boid agent</description>
11 ⊞     <memory>
41 ⊞     <functions>
68 ⊞     <states>
74       <gpu:type>continuous</gpu:type>
75       <gpu:bufferSize>2048</gpu:bufferSize>
76     </gpu:xagent>
77   </xagents>
78 ⊞ <messages>
124 ⊞ <layers>
136 </gpu:xmodel>
```

Figure 3.4: The Boids model description within the 'XMLModelFile.xml' file.

3.3.2.1 Memory

Agent memory refers to the persistent variables that an agent stores throughout its simulation lifetime. Agent memory consists of a number of variables used

```
<gpu:xagent>
  <name>Boid</name>
  <description>a simple boid agent</description>
  <memory>
    <gpu:variable><type>int</type><name>id</name></gpu:variable>
    <gpu:variable><type>float</type><name>x</name></gpu:variable>
    <gpu:variable><type>float</type><name>y</name></gpu:variable>
    <gpu:variable><type>float</type><name>z</name></gpu:variable>
    <gpu:variable><type>float</type><name>fx</name></gpu:variable>
    <gpu:variable><type>float</type><name>fy</name></gpu:variable>
    <gpu:variable><type>float</type><name>fz</name></gpu:variable>
  </memory>
```

Figure 3.5: Memory variables (Boids model).

to store agent information, such as position, velocity, etc. These variables are strictly typed using C basic data types. The Boids model contains the variables that describe each agent's location (`x, y, z`) and velocity (`fx , fy, fz`), as well as an identifier (`id`). Figure.3.5 shows a screenshot of all Boids model memory variables.

3.3.2.2 States

An agent has at least one state. States are used to distinguish between agents of the same form that may have different life-cycle functional capacities. A biological cell agent, for example, may be in a state of normal activity or may be in the division or dying phase. Having different representations of agents in different states ensures that complex systems exhibiting heterogeneous behaviour can be represented by groups of homogeneous behaviours. This helps to avoid a divergence within behavioural scripts, a problem known to negatively affect GPUs' performance. Within the Boids model, only a single state 'default' is required, as all agents within this model perform the same homogeneous behaviour throughout the simulation, as shown in Fig.3.6.

```
<states>
  <gpu:state>
    <name>default</name>
  </gpu:state>
  <initialState>default</initialState>
</states>
```

Figure 3.6: States list (Boids model).

3.3.2.3 Functions

The optional agent function list within the X-machine agent representation must contain at least one agent function element. The function describe the behaviour script that dictates how an agent updates its internal memory. In order to provide information about the functional operation, a description of the scripted function must be provided in the model file. The function definition must also include a non-optional name and optional description, a current state, next state, an optional (single message input, single message output, single agent output, global function condition, function condition), a reallocation flag and a random number generator flag [122, 124]. The structure of the agent functions list is shown in Fig.3.7 and consists of all optional and non-optional elements.

The `currentState` value is used to filter agents by only applying the function to agents in the specified state. Once the agents complete their function, they move into the next state that appeared within `nextState` the element. Both the `current` and `nextState` values must exist as states in the state list (states) definition. The reallocate agent is an indication that the agent performing the agent function may die as a result and need to be removed from the population. The reallocate element can be either true or false. By default, this element is true to perform removing dead agents automatically; however, when 'false' is specified, removing dead agents is not performed, even if an agent indicates that it has died. The RNG tag is used to indicate whether a random number generation is required (RNG). If the flag is set to true, an additional parameter is passed to the agent function within the Function file, which contains a number of seeds used to produce parallel random numbers[122].

Agent function message outputs and inputs: The inputs and outputs are defined as a set of variables in XMML in the form of a named message type. The message inputs of the agent function means that the agent function will iterate through the list of messages with a name equal to the `messageName` element specified within the function description. Within the Boids model, message inputs can be found in a function called `Inputdata` as seen in Fig 3.8. The `outputs` element in an agent function generates messages for other agents to read on the message board. The outputs element includes the `messageName` and `type`. The `messageName` element has therefore already been identified

```
<functions>
    <gpu:function>
        <name>func_name</name>
        <description>function description</description>
        <currentState>state1</currentState>
        <nextState>state2</nextState>
        <inputs>...</inputs>                                  //optional
        <outputs>...</outputs>                                //optional
        <xagentOutputs></xagentOutputs>                       //optional
        <gpu:globalCondition>...</gpu:globalCondition> //optional
        <condition>...</condition>                            //optional
        <gpu:reallocate>true</gpu:reallocate>                 //optional
        <gpu:RNG>true</gpu:RNG>                               //optional
    </gpu:function>
</functions>
```

Figure 3.7: The definition of an agent functions list within the model description file. The source of the code [122].

by an XMML file. A message can't be an output without an actual message specified within the XMML model file. The `type` element can either be a single-message or optional-message, wherein a single-message means that only one message is output by each agent executing the function, while the optional-message allows for either a single-message or no message to be output [122]. An example of the implementation of message output within the Boids model is shown in Fig.3.9.

```
<gpu:function>
  <name>inputdata</name>
  <currentState>default</currentState>
  <nextState>default</nextState>
  <inputs>
    <gpu:input>
      <messageName>location</messageName>
    </gpu:input>
  </inputs>
  <gpu:reallocate>false</gpu:reallocate>
  <gpu:RNG>false</gpu:RNG>
</gpu:function>
```

Figure 3.8: An example of message inputs (inputdata function) within the Boids model.

Agent function X-Agent outputs: The birth of an agent during the simulation is handled by the `xagentOutputs` element. This consists of non-optional `xagentName` and `state` elements. The name of the newly added agent is equal to the value of the `xagentName` element, and the state of this agent is the same as the one specified within the `state` element. Both values must already be defined within the model specification file. The development of new discrete agents is not permitted for continuous type agents (which must also be continuous type agents) [122].

```
<gpu:function>
  <name>outputdata</name>
  <currentState>default</currentState>
  <nextState>default</nextState>
  <outputs>
    <gpu:output>
      <messageName>location</messageName>
      <gpu:type>single_message</gpu:type>
    </gpu:output>
  </outputs>
  <gpu:reallocate>false</gpu:reallocate>
  <gpu:RNG>false</gpu:RNG>
</gpu:function>
```

Figure 3.9: An example of message outputs (outputdata function) within the Boids model.

function conditions:As a mechanism for transiting sub-sets of agents into various states, functions may have conditions. A function condition means that the function must be applied to agents to meet the condition (and in the correct state stated in the `currentState`). For example, a function that simulates an agents death is applied only to the agents meeting the condition to move into a dead state. Within a function condition, there are three main components: a left-hand side statement, an operator and a right hand-side statement. Both `lhs` and `rhs` statements may contain either an `agentVariable` (defined within agent memory), or constant values or recursive conditions, as shown in Fig.3.10. The function condition in this example performs the following pseudo code: (variable_name1) ((variable_name2)+(1)).

```
<condition>
    <lhs>
        <agentVariable>variable_name1</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
        <condition>
            <lhs>
                <agentVariable>variable_name2</agentVariable>
            </lhs>
            <operator>+</operator>
            <rhs>
                <value>1</value>
            </rhs>
        </condition>
    </rhs>
</condition>
```

Figure 3.10: Example of a function condition with a recursive condition element.

Global function condition: A global function condition is similar to a

function condition in its syntax, but acts globally to determine whether the function should be applied to either all the agents or none of them (within the correct state in the `currentState`). This can be implemented by using both values attached to the `maxItterations` and `mustEvaluateTo` elements. The `maxItterations` element value determines the number of iterations that the function can be executed before applying the global condition and the `mustEvaluateTo` element to determine whether the global function can be applied to all the agents or not based on the stated value. If the `mustEvaluateTo` value is equal to `true`, the function is applied to all the agents, whereas if the value is `false`, the function cannot be applied to any of the agents. An example of a global function condition is shown in Fig.3.11. The definition in this figure is the result of the pseudo code condition: (((movement) (0.25)) == true). The global condition will be ignored before it reaches 200 iterations, after which the function will be applied to every agent.[122]

```
<gpu:globalCondition>
    <lhs>
        <agentVariable>movement</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
        <value>0.25</value>
    </rhs>
    <gpu:maxItterations>200</gpu:maxItterations>
    <gpu:mustEvaluateTo>true</gpu:mustEvaluateTo>
</gpu:globalCondition>
```

Figure 3.11: An example of a global function condition [122].

3.3.2.4 Messages

Messages enable agents to communicate with each other. Agents information can be passed as variables of messages and are stored in the memory board located in the global GPU memory, which is accessible to all agents. Defining agent messages within model specification files consist of a non-optional `name`, or an optional, `description` of the message, a list of variables, a `partitioningType` and a `bufferSize`. The `bufferSize` element is used to define the maximum number of this message type that exists within the simulation. For the `partitioningType` element, as mentioned in section 3.2.3, there are three partitioning techniques supported by FLAME GPU to ensure that message variables are provided to agents in an optimal manner for

processing. These techniques include non-partitioned (`partitioningNone`), discrete 2D space partitioning (`partitioningDiscrete`) and 2D/3D spatially partitioned space (`partitioningSpatial`). Furthermore, within the Boids model, as mentioned before, there is only one message called `location` Fig.3.12 shows the agent messages description of the Boids model.

```
<messages>
  <gpu:message>
    <name>location</name>
    <description>a message holding the location of an agent</description>
    <variables>
      <gpu:variable><type>int</type><name>id</name> </gpu:variable>
      <gpu:variable><type>float</type><name>x</name></gpu:variable>
      <gpu:variable><type>float</type><name>y</name></gpu:variable>
      <gpu:variable><type>float</type><name>z</name></gpu:variable>
      <gpu:variable><type>float</type><name>fx</name></gpu:variable>
      <gpu:variable><type>float</type><name>fy</name></gpu:variable>
      <gpu:variable><type>float</type><name>fz</name></gpu:variable>
    </variables>
    <gpu:partitioningSpatial>...</gpu:partitioningSpatial>
    <gpu:bufferSize>2048</gpu:bufferSize>
  </gpu:message>
</messages>
<layers>
```

Figure 3.12: Message description of the Boids model within the model specification file.

3.3.2.5 Function Layers

The model execution layers should include all defined agent functions within the model. The function layers represent the control flow of the simulation processes. The complete execution of all agent function layers produce a single simulation iteration that can be repeated any number of times. In FLAME GPU, the user needs to describe the sequential order of the agent's functions using function layers within the XMML Model File. Based on the order of layers the agent functions are executed. Each function layer can contain any number of functions from different agents. The functions within a layer must have no dependencies on each other and may only depend on outcomes generated by the previous layer. If functions have no dependencies, they may be placed in the initial layer. Functions that depend on some earlier processing of the same agent must be placed in a subsequent layer (internal dependency). Functions that depend on some earlier processing in another agent must also be placed in a subsequent layer (communication dependency). Within the Boids model, as shown in Fig. 3.13, two types of dependencies force the model functions to be

in separate layers. There is, however, a dependency between the `outputdata` function and the input data function via message location. The second type of dependency is based on computation, as can be seen between the input data function and the move function. The move function is dependant on the computation of the velocity vectors carried out by the input data function and is stored within the agent's internal memory[122, 124].

```
<layers>
  <layer>
    <gpu:layerFunction>
      <name>outputdata</name>
    </gpu:layerFunction>
  </layer>
  <layer>
    <gpu:layerFunction>
      <name>inputdata</name>
    </gpu:layerFunction>
  </layer>
  <layer>
    <gpu:layerFunction>
      <name>move</name>
    </gpu:layerFunction>
  </layer>
</layers>
```

Figure 3.13: Function layers of the Boids model within the model specification file.

3.3.3 Model Behaviour(Agent Function Scripts)

Agent functions are a mapping (or updating) of the agents' internal memory; this occurs when an agent moves from one state to another. Agent function scripts containing agents' behaviours must be defined within a suitable script within the function.c file after adding the function description to the model XML file. The agent functions' behaviour is described from a single agent's perspective, but the simulator will apply the same agent function code to each agent in the correct starting state in parallel (and applies any of the defined function conditions). Agent function scripts are described using a simple C-based syntax with agent function statements, and the function arguments depend on the XMML function definition. The declaration of all FLAME GPU functions have a prefix, __FLAME_GPU_FUNC___, followed by the name and the set of function arguments. The name of the function must be exactly the same as in the XMML template file. Agent and message data structures are dynamically created and defined in the header file 'header.h'. The data

of individual agents are passed as the first argument in the form of a structure (xmachine_memory_agent_name), where the agent_name refers to the agent name defined in the XMML model file. The internal memory variables of agents are updated by changing the members of this structure. Fig.3.14 shows the (xmachine_memory_agent_Boid) structure that is generated from the Boids memory, as defined in Fig.3.5. Message variables are also included within a structure named xmachine_message_message_name. The structure of arrays that hold the agent and message list have also been included within the header file. The xmachine_memory_agent_name_list structure for the agent and message list is under the name of the xmachine_message_message_name_list structure.

```
struct __align__(16) xmachine_memory_Boid
{
    int id;    /**< X-machine memory variable id of type int.*/
    float x;    /**< X-machine memory variable x of type float.*/
    float y;    /**< X-machine memory variable y of type float.*/
    float z;    /**< X-machine memory variable z of type float.*/
    float fx;    /**< X-machine memory variable fx of type float.*/
    float fy;    /**< X-machine memory variable fy of type float.*/
    float fz;    /**< X-machine memory variable fz of type float.*/
};
```

Figure 3.14: Code snippet of automatically generated Boid data structures within 'header.h' file.

Since agents can affect and be affected by their neighbours, they also have a simulation API that allows an agent to access their neighbours message variables. There are three basic steps in the process of accessing message variables. First, a pointer has to be identified within an agent function to allow the data structure with message variables to be accessed. Second, by the API function get_first_message_name_message (arguments,..), an array with the first group of messages are loaded into shared memory. Finally, within a while loop, the simulation API function get_next_message_name_message (arguments,..), which reads messages sequentially in shared memory, is called until the first group of messages in the shared memory is exhausted. The get next message name message function loads the next group of messages into the shared memory after the first group message list is exhausted. Fig.3.15 shows how to use the get_first_location_message and the get_next_location_message functions within the inputdata function Boids model. The existence of the xmachine_message_location_list

argument within the function definition as a result of the XMML model file indicates that this function inputs the location message.

```
__FLAME_GPU_FUNC__ int inputdata(xmachine_memory_Boid* xmemory,
xmachine_message_location_list*location_messages)
{
xmachine_message_location* location_message;
location_message = get_first_location_message(location_messages);
while(location_message){
//check for messages from self
if (location_message->id != xmemory->id){
//check distance between message and self
float separation = distance(xmemory -
location_message);
if (separation < INTERACTION_RADIUS){
//update perceived global centre
//update average velocity
//update avoidance vector
...
}
}
location_message = get_next_location_message(
location_message, location_messages);
}
//update the velocity vx, vy, and vz
...
return 0;
}
```

Figure 3.15: The code of the InputData Function within the Boids model.

The `outputdata` function in Fig. 3.16 uses a dynamically generated function called `add_location_message`. As the definition in the XMML model file states that it will output a message of type location, the agent function can call upon the `add_location_message`. A value of 0 is returned by the agent to announce that it is still alive. Any non-zero value indicates that the agent is dead and should be removed from the simulation.

```
__FLAME_GPU_FUNC__ int outputdata(xmachine_memory_Boid* xmemory,
xmachine_message_location_list*location_messages)
{
add_location_message(location_messages,
xmemory->id,
xmemory->x,
xmemory->y,
xmemory->z,
xmemory->fx,
xmemory->fy,
xmemory->fz);
return 0;
}
```

Figure 3.16: The code of the OutputData Function within the Boids model.

3.3.4 FLAME GPU Template Files

The FLAME GPU SDK has a number of XSLT templates that are linked to function script files to generate a dynamic simulation code. These templates can be summarised as follows:

- `header.xslt` This template generates the header file that consists of any agent and message data structures and creates function prototypes for simulation functions.
- `main.xslt` This template is responsible for producing a file that can define the main function to handle the initialisation of the GPU device and deal with the command-line options.
- `io.xslt` The source file that is generated by this template contains functions for loading initial agent XML data into the simulation and saving the state of the simulation in XML format.
- `simulation.xslt` The source file that is generated by this template contains the host side simulation code.
- `FLAMEGPU\_kernels.xslt` This template file produces a CUDA-header file, which includes the CUDA kernels and device functions.
- `visualisation.xslt` This template is responsible for generating a source file that allows for the basic visualisation of the simulation.

FLAME GPU generates simulations by applying these templates to the model files, which are linked with the behaviour scripts to generate a simulation program. All agent and message memory will be accessed during this process using fast caches, shared memory for agent variables and texture memory for message variables.

3.3.5 Model Execution and Visualisation

FLAME GPU simulations require a variety of arguments based on either console or visualisation mode requirements. In both cases, the first argument is always the file location of the initial agent XML file containing the initial agent data. The agent's initial variable values should be contained in tags using the name of the variable. An example of the Boid agent taken from the initial agent file

```
<xagent>
<name>Boid</name>
<id>1</id>
<x>0.4107396</x>
<y>0.09933436</y>
<z>0.18894494</z>
<fx>-0.048779726</fx>
<fy>0.021190405</fy>
<fz>0.3741398</fz>
</xagent>
```

Figure 3.17: An initial state of a Boid agent taken from an initial agent XML file as an argument.

is shown in Fig.3.17. It is also possible to pass extra optional CUDA arguments (i.e. device = 1) to set the CUDA enabled GPU device. The default value for the argument for CUDA devices is 0. To specify the number of iterations within the console mode, an additional argument is required. After running the simulation, a number of XML output files are generated (where they are in the same location of the input file). Output files can be used as inputs for later runs or checking points, especially when it comes to validating the results. Statistical information about the simulation can be collected by writing scripts that parse the agent output files. For example, a Python script can be specified to iterate over the output files and plot each agent's average distance between them over time. There are various macros within the FLAME GPU in the generated simulation header file (header.h) that can be modified to output extra information about the population, such as more detailed timing results or population count per iteration. Running the simulation using the visualisation mode only needs the initial agent data file, and the number of iterations is not required as the simulation will continue to run until the user closes the visualisation window [124].

3.4 Summary

FLAME GPU is a parallel, agent-based simulation framework that enables real-time model interaction and visualisation via the GPU. This chapter demonstrated how a simple multi-agent system (Boids model) is modelled using FLAME GPU. This chapter also showed the features and capabilities of FLAME GPU. However, a number of issues may negatively affect the performance of FLAME GPU, as the GPU has limited memory resources. There are no specified limits on the maximum number of agent and message variables; however,

an increased number of variables (memory) will harm the overall performance. Based on the formal definition of a communicating X-machine, function transitions between states requires updating all agent and message variables in the memory. Updating a subset of agent and massage memory will reduce memory usage. The following chapters propose a new approach (Data aware approach) that allows FLAME GPU to run simulations with less memory movement. This approach may enhance the overall performance and allow a larger population using the same GPU device.

In FLAME GPU, function layers represent the control flow of the simulation. All functions in the same layer are executed in parallel. Functions of the same agent and those that have communication dependencies must be separated in different layers in order for the executions to be sequential. The complete execution of all function layers represents a single simulation iteration. Reducing the number of layers will reduce the time it takes to run the single iteration. Later in this thesis, an approach (Functional approach) will be proposed to discuss this matter.

Chapter 4

Methods and Experimental Plan

4.1 Introduction

This chapter describes the methods used to address the thesis goals stated in the first chapter. It sets the rules for the benchmark model and describes how to build the model generator to produce complex models following the same rules. It also includes the techniques used to design, evaluate, and validate both approaches using data dependency to enhance FLAME GPU performance.

4.2 Designing the Benchmark Model

The first focus in our work is on creating a new benchmark model that can easily measure the system's ability to deal with each of the following elements: system scalability, system homogeneity, and the ability to handle increases in the level of agent communication. A description of the proposed model can be found in the next chapter. This chapter presents the simple rules to follow during its implementation. In the first stage, the design is focused on creating a model from scratch consisting of two types of molecules (e.g. A and B). A simple reaction will occur when one A molecule interacts with one B molecule to create a C molecule, assuming that A+B=C represents the relationship between the three molecules. The simple formula syntax consists of at least two molecules, a (+) operator between them, an equals sign, and a third type of molecule (A+B=C). Assuming that A is the master agent, B is the slave agent, and C is the combined agent. It is named this way to differentiate its functions within the simulation, and it is a convenient way to construct our model. Converting the formula syntax to moving agents requires a set of rules for each type of

agent. The following points show the basic rules for each agent type to create the simulation using FLAME GPU:

- The first molecule in the formula (equation) is the master agent. All the molecules after the master agent and before the equals sign are classified as slave agents. The molecule after the equals sign will be the combined agent.

- The master agent consists of a set of variables presenting the agent memory, five functions to describe agent behaviour, and two states (moving and dead), as shown in Table 5.1. Only one master agent can be found within one equation. The master agent can communicate with every slave agent within the same equation.

- The slave agent consists of a set of variables presenting the agent memory, four functions to describe agent behaviour, and two states (moving and dead). The slave agent can be one agent or more within the same equation, and it only communicates with the master agent.

- Every slave agent type requires three types of messages to communicate with the master agent, and that will be multiplied every time an extra slave agent is added to the model.

- The combined agent consists of a set of variables presenting the agent memory, one function to describe agent behaviour, and one state (moving). There is one combined agent type for each slave agent that interacts with the master agent.

- The movement speed and radius interaction value can be used as model parameters to control agent behaviour.

4.2.1 Designing FLAME GPU Generator

The motivation for designing and building this tool is to produce complex models based on the simple model that has been designed. The complexity that we are aiming to achieve is the ability to measure both system scalability and system homogeneity, which cannot be done using the simple reaction of two molecules. The design of the generator allows the benchmark model to increase divergence within agents and populations.

The model generator is an object-oriented program written in c. It has been designed to accept any number of equations, parse them, and generate FLAME GPU model. Three files that can be generated by the model generator, which is required to run the simulation using FLAME GPU. Figure 4.1, show how the generator parses the input syntax.

Figure 4.1: Algorithm: parsing the formula syntax and generate FLAME GPU model

Input: Formula syntax within text file

Result: XMLModelFile.xml, Functions.c. InitialAgentData.XML

```
1  Read in the files and create a FileModel object
2      for each unique agent name in a vector
3            create a FLAMEGPU Agent
4              add any agent variables and functions that are the same as
     master/slave/combined
5      If agent is ONLY a combined
6    then add its functions (and variables)
7     If agent is a slave
8        then add the functions (and variables/messages) required to bind
     with a master
9     If agent is a master
10     then add the functions (and variables/messages) required to bind with
11       ALL of the slaves.
12
13   add FLAMEGPU agent to FLAMEGPUModel
14   Open a file
15   call FLAMEGPUModel outputXML(file )
```

4.2.2 Testing system scalability

To measure the ability of the system to scale and handle the large number of populations, we used a simple model representing A+B=C produced by the model generator to generate a new initial agent data file with extra populations. To measure scalability, in all experiments conducted in this research the population starts with 100000 agents for each type and ends with 800000.

4.2.3 Increasing Agent complexity

The model generator is able to parse equations that consist of more than one slave agent. Adding an extra slave agent to the equation increases the master agent functions, which means an extra function layer is needed to execute and an extra massage list will be iterated by the master agent. This experiment

increases divergence within the agent and helps to measure the ability of the system to deal with this type of problem.

4.2.4 Increasing Population complexity

The model generator helps to increase the divergence within the population by parsing more than one line of the equation. Every line consists of one formula, and each formula has a different type of molecule. This experiment will increase the number of functions within each function layer.

4.3 The Discovery of Data Dependency

In the beginning of the research, data dependency between agent functions was manually investigated in the benchmark model. The extracted dependency information was subsequently used to apply new approaches for minimising data movement and thereby enhancing system performance. This investigation was then further extended to automate the discovery of data and message dependencies. One of the aims of this thesis is to design and implement automated dependency parsing using Flex and Bison tools. The following section describes the process of building the dependency parsing tool.

4.4 Compiler Construction

Programming languages are the route for communication between people and computers, and a compiler translates a high-level programming language into a low-level language [3]. There are different types of compilers. The most common type converts high-level program code into intermediate-level assembly language or low-level machine language. Converting intermediate code into machine language is done by a type of compiler called an assembler. A trans-compiler, or a source-to-source compiler, is another compiler type that converts a high-level programming language to another programming language at the same level.

The modern compiler structure consists of two phases: front-end and back-end, as shown in Fig 4.2. The front-end phase consists of lexical analysis, syntax analysis, and semantic analysis, while the back-end phase consists of intermediate code generation, code optimisation, and target code generation.

In our implementation of dependency parsing, we are interested in the front-end phase of the compiler. The following sections will discuss the stages of the front-end phase in detail.

4.4.1 Flex and Bison

Building a compiler program from scratch is a complicated task. Using existing compiler-construction tools will help to minimise and hide some details of the generation algorithm [3]. Examples of these tools are scanner generators and parser generators. Flex and Bison are free tools developed by the GNU Project[1]. Flex is the fastest scanner generator, and Bison is a parser generator. Flex and Bison can be connected to build a compiler, and they can also work separately, depending on the software required. Flex and Bison have a large and active community, and there are a large number of projects that have been designed using these tools. There are many details available about how these tools work and help to build interpreters and compilers. These tools are a good choice for implementing dependency parsing.

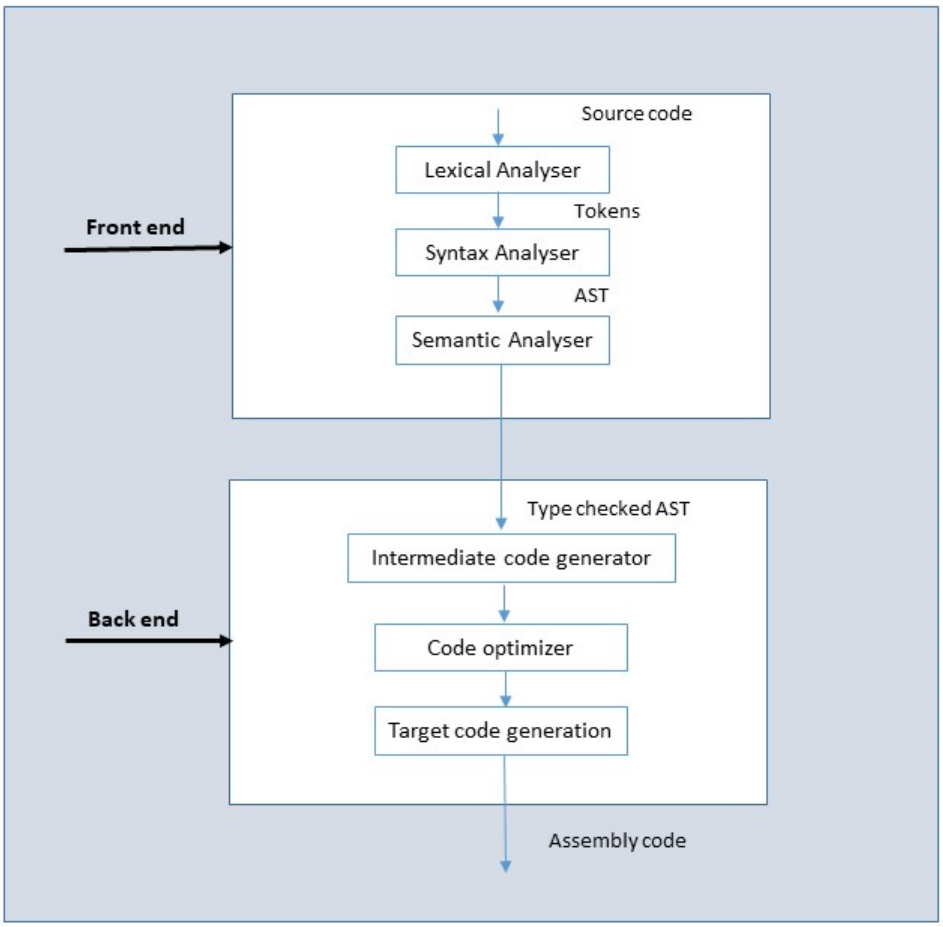

Figure 4.2: Modern compiler phases

The Flex program is used to scan the source code. Flex (Fast LEXical analyser generator) is used to scan and tokenise the input. Depending on the predefined rules, the list of tokens will be different. So, in this case, a file of rules will be supplied that supports the C language and all keywords that may be included within agent function scripts.

The parsing stage also needs a predefined grammar for C syntax. The Bison program (the parser to be used in this system) will be linked to Flex. The parser will get the results of low-level parsing (tokens) from the first stage to produce the parse tree. The parsing stage has general functions that can be used by all parsing files (the extra program code that is linked to the parser, which performs further analysis). These functions contain operations to optimise and generate the final result. Fig 4.3 shows a summary of the system structure. This is explained in more detail in the following sections.

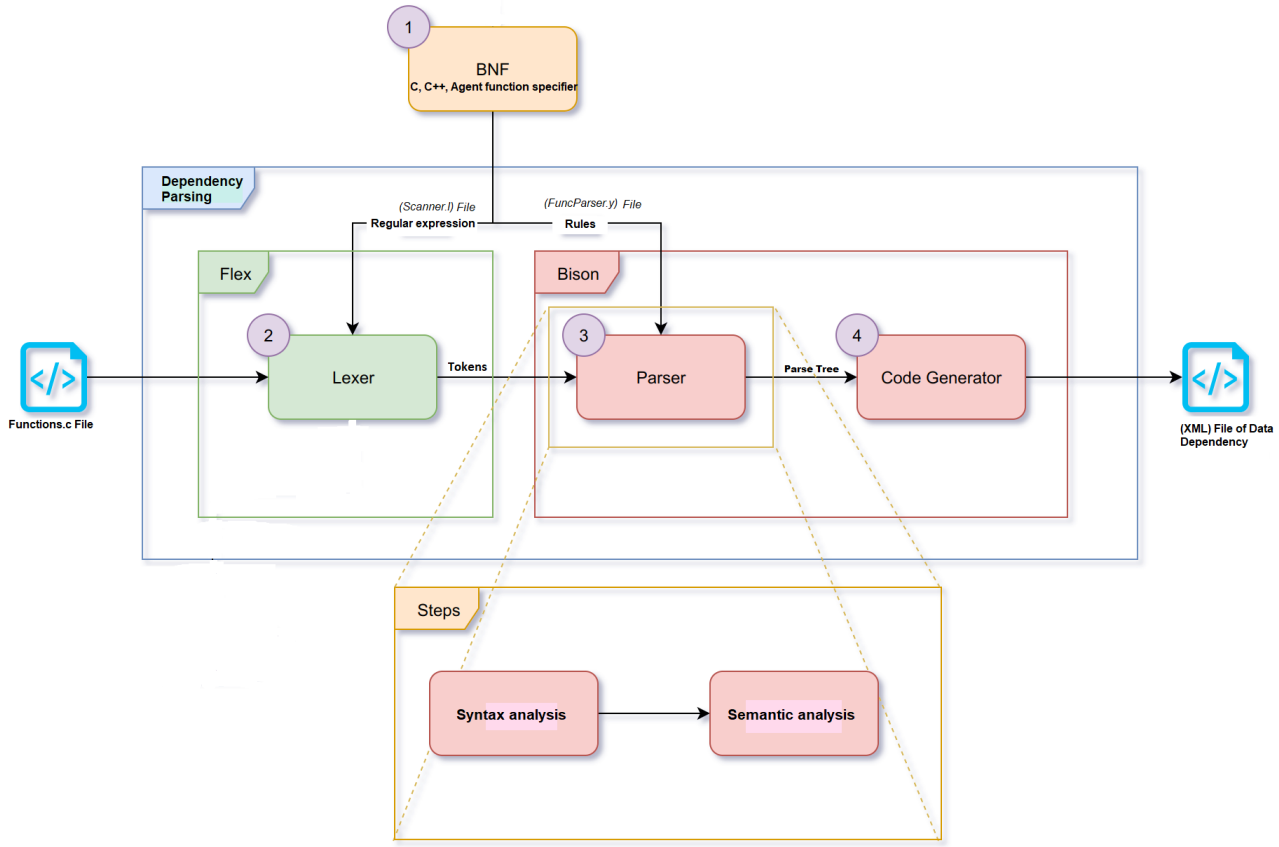

Figure 4.3: The dependency parsing system

[1]http://www.gnu.org/software/bison/

4.5 The Scanner

Scanning, or lexical analysis, is the first step in the front-end of the compiler phase. The scanner generated by Flex matches the input character stream against one of the token patterns defined in the regular expression file (that was used by Flex to generate the scanner). It presents a syntax error for any word or character that does not match any of the given patterns. Figure 4.4 shows how the scanner works in a simple way. During this stage, the C code is converted to a list of tokens. C code, as with any programming language, consists of a number of tokens.

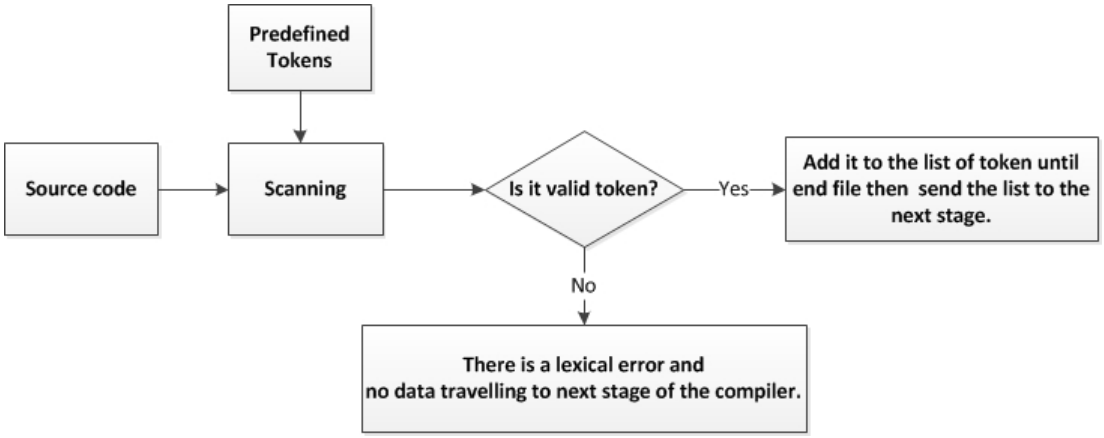

Figure 4.4: Lexical analysis process

The input to the Flex is generally a file with a ".l" extension, which contains regular expressions for the language. This file needs a specific structure, as shown below, to work correctly. It consists of three sections separated by two percent symbols, as shown in the listing 4.1. The first one is a section of declarations and option settings. It includes all header files, variables, and function declarations that will be used in the C code section and the C definition section. The rules section contains the list of regular expressions of the language that need to be analysed. With each rule, there is an action to be applied to matches; for example, *(if you find this operator return this value)*. The last section is the main part of this file, which contains the call for the main Flex function `yylex()`. This function uses simulated finite-state machines to identify each token and then return the numeric code of their types to the parser. The function yylex() is called by the parser directly to obtain tokens for parsing.

```
1 %{
```

```
C declarations
%}
flex Definition section
%%
Rules section
%%
C code section
```

Listing 4.1: The Flex input file structure

Flex also includes a number of functions that work with `yylex()` to generate the list of tokens. After Flex reads the input file, it generates the `yy.lex.c` file. Compiling this file using the C compiler will generate an output file `a.out`. Executing the `a.out` file with a specified input file will analyse the input stream and produce the list of tokens. Thus, in the end, this output file is the actual scanner, as shown in Fig 4.5[2].

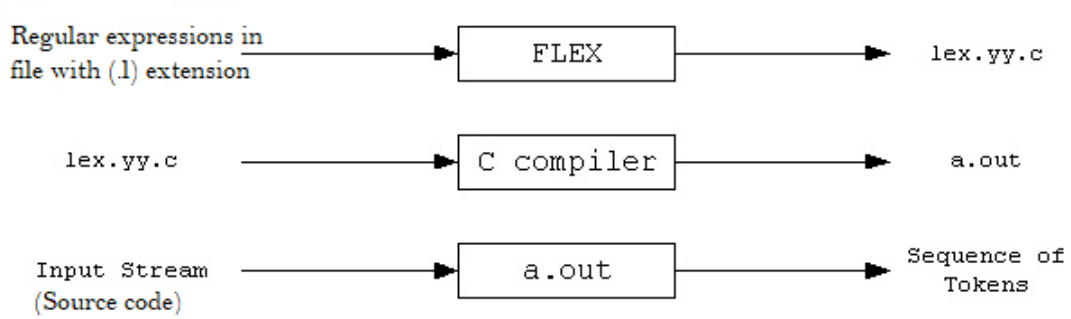

Figure 4.5: Steps for generating a scanner using Flex.

4.6 The Parser

Parsing is an important design step for the compiler. It requires a context-free grammar (CFG) to generate the programming language that needs to be parsed. The main goal of this stage is to make sure that the given list of tokens is compatible with the language's CFG. Parsing, or syntax analysis, will build the syntax tree if all of the given tokens match with the rules. If any of these tokens do not match with the rules, it will present a syntax error.

To understand the parsing process, a further explanation of the CFG is needed, including an explanation of how to build a syntax tree using these rules. A CFG is defined as a number of rules that are used to produce patterns

[2]http://alumni.cs.ucr.edu/ lgao/teaching/flex.html

of tokens that help to define the syntax tree of the language [3]. It consists of four components:

- **Terminals** The terminals in the grammar are the leaves of the parse tree, and they can include digits or longer scanned tokens returned by the lexer. *X, 9, and 1* shown in Figure 4.6 are examples of terminals.
- **Nonterminals** are the branch nodes in the parse tree. Nonterminal symbols are used to write every grammar rule.
- **Start symbol** is the special nonterminal symbol that serves as the starting point for the rules and corresponds to the root node of the parse tree.
- **Productions** are the rules that were organised in a specific way to build the grammar for the programming language.

Code listing 4.2 displays an example of a CFG for the `IF` statement. This example is used to clarify the idea of a CFG and to demonstrate the expected syntax tree. The start symbol of this CFG example is `Ifstmt`; `IDEN, NUM, OP` are the terminal symbols that refer to different kinds of tokens; `IDEN` refers to identifiers; `NUM` refers to integer literals; `OP` refers to operators; and the nonterminal symbols of these rules are `stmt, exp`.

```
1 ifstmt-> if ( exp ) stmt
2 stmt  -> exp
3 exp   -> exp OP exp
4 exp   -> IDEN | NUM
5 OP    -> +|-|*|/|<|=
6 IDEN  ->A|...|Z|a|...|z
7 NUM  ->0|...|9
```

Listing 4.2: Analysing large textual units

Using this CFG example will produce the syntax tree for any example of an `IF` statement. Figure 4.6 represents (`if(x<9)x=x-1`) as a parse tree generated by the CFG shown in listing 6.3.

The Bison tool works as a parser in this phase to generate both syntax and semantic analysis. Bison requires an input file with a `(.y)` extension that consists of the CFG rules. The structure of this input file is similar to the Flex

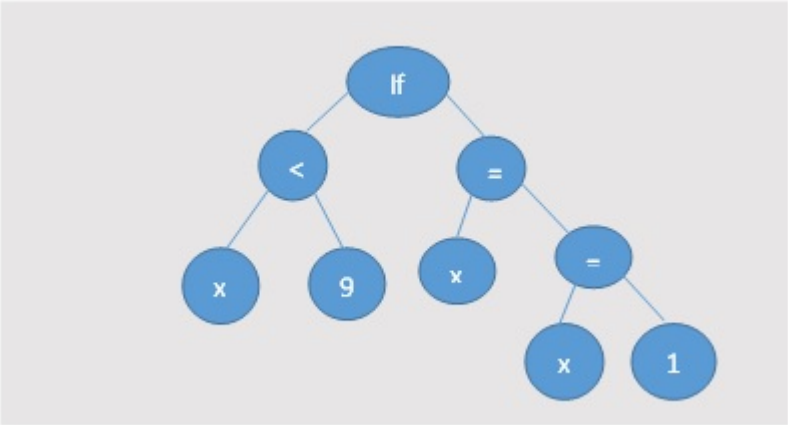

Figure 4.6: Syntax Tree for simple *If* statement

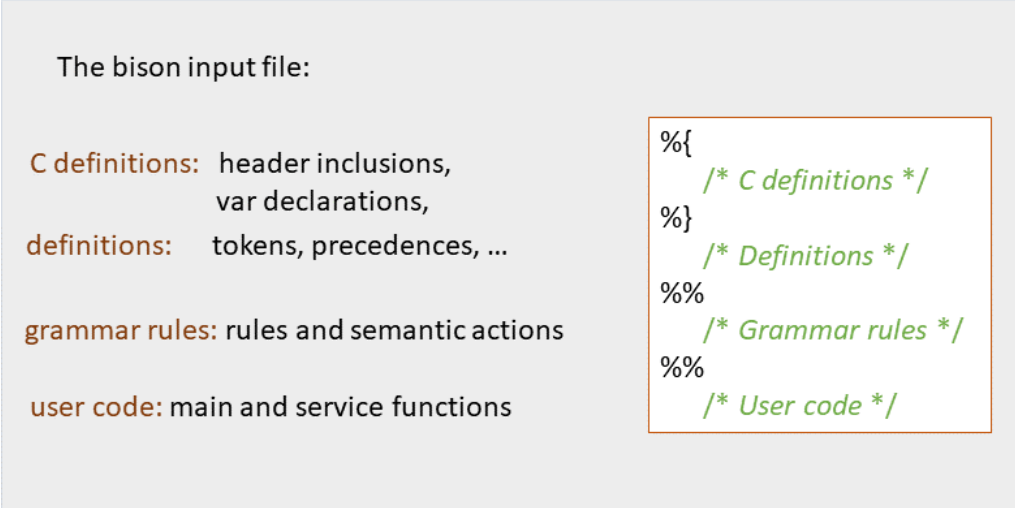

Figure 4.7: Bison input file structure

input file, as shown in Fig 4.7. Bison generates a file named *parser.tab.c* after reading the input file. Compiling the *parser.tab.c* file using any C compiler will produce an executable file. This executable file is the parser, which is run with the input file to build the syntax tree.

Semantic analysis will occur after syntax analysis, creating the abstract syntax tree (AST) and the symbol table. This process will use both the AST and symbol table to check the meaning of the line of the grammar; in addition, it will check the type of symbol and determine its scope of use. It does this through linked functions triggered during the parsing process. The implementation of both stages to create the FLAME GPU model generator will be detailed in Chapter 7

4.7 Research Validation

To validate the results of running the benchmark model using FLAME GPU three functions have been included within the functional scripts to calculate information about the simulation. These functions are user-defined functions and need to be defined within the model specification XML file. The first function is called the initialisation function and used to set constant global variables. It will be called a single time automatically once the FLAME GPU generates the simulation code. The second function is called a step function, and it is called at the end of each iteration to calculate agent account. The third function called exit function, and it will be called by the end of the whole simulation to print out each agent type's final population size. The output results will be used to examine the simulation validation and verification. The body structure of these functions can be found within the function.c file in appendix A and figure 5.2 in Chapter 5 is an example of the output results of these functions.

Chapter 5

Benchmarking Agent Based Modelling systems

In most research areas, benchmarking is the quantitative basis of the software or hardware devices created. The purpose of assessment is to demonstrate the applicability of an approach under certain constraints and to provide decision-making support for choosing the best approach to a certain problem. Benchmarking, including a comparative assessment of various methods, is one way to evaluate these issues [163]. The growing number of agent-based applications in the simulation and AI fields has led to an increase in the number of studies focused on evaluating the modelling capabilities of these applications. Observing system performance and how applications behave during increases in population size is the main focus of benchmarking in most of these studies. However, system scalability is not the only issue that may affect the overall performance of an ABM. This chapter presents a new benchmark model and reports on its performance characteristics within the FLAME GPU simulator as an example of a parallel framework for ABM. The aim of this model is to provide parameters to easily measure the following elements: system scalability, system homogeneity, level of communication and model complexity. The proposed benchmark can provide insight into the overall performance characteristics of specific ABM frameworks.

5.1 Benchmarking ABM criteria

As mentioned in Chapter Two, there are a number of popular agent-based modelling and simulation frameworks that are used to build models. However, the

scalability and performance limitations in these systems prevent modellers from simulating complex systems at very large scales. This is because many of these frameworks were designed to be run on a single CPU architecture, and as a result they cannot perform the necessary computation (within a reasonable time) of a large number of agents. For this performance reason, a number of platforms and simulators have arisen to deal with large-scale simulation. Repast HPC[1], D-Mason[2]and FLAME GPU[3]are examples of these kinds of platforms, which use parallel and distributed approaches to run large-scale (HPC) simulations.

There have been several studies reporting on computational performance in most ABM frameworks [42, 14, 83] for specific models. Varying the population size to measure system scalability is the most common benchmark. A benchmarking process is an excellent way to determine the characteristics of simulator performance, but unfortunately there is no standard method to benchmark ABM that considers characteristics of the model beyond scale. Thus, there is a need to design a benchmark model that considers characteristics of an ABM that affect performance. The OpenAB community [4]summarised a number of criteria that may affect this performance:

- Arithmetic intensity: the computational complexity of an agent or population.
- Scale: varying population size.
- Model memory: the internal memory requirements of an agent or population.
- Inter-connectivity: the level of communication between agents.
- Homogeneity: divergence of behaviour within an agent or population.

5.2 Benchmarking ABM models (background review)

Numerous ABMs have been used to address a number of issues, such as testing and analysing simulation tools and comparing ABM platforms, and they have

[1]https://repast.github.io/repast_hpc.html
[2]https://sites.google.com/site/distributedmason/.
[3]http://www.flamegpu.com/
[4]http://www.openab.org/.

been used as teaching tools for modelling real systems. This section reviews some of these models and their purposes.

Railsback et al. [116] proposed a simple model called StupidModel that can be easily implemented on any ABM platform. This model contains a number of versions to increase simulation complexity, starting from moving agents to a full predator–prey model. StupidModel was developed to be a teaching model for ABM platforms, such as NetLogo and Swarm. It is also used as a benchmark model to compare modelling capabilities and performance between several ABM platforms [83, 117, 84, 139].

Predator–prey is the most commonly used model in the field of ABM and simulation. Developed by Alfred Lotka (1925) and Vito Volterra (1926), it is based on two differential equations and describes the dynamics of predator–prey behaviour. The basic rules of predator–prey in ABM can be summarised as follows: 1) two types of populations represent prey and predator agents; 2) the prey population will increase by moving to food resources and decrease by being eaten by the predators; 3) the predator population will increase by eating the prey and will decrease by starvation; and 4) both populations are moving randomly and following simple rules to communicate with the environment and with each other.

Several studies have compared the execution efficiencies of different ABM platforms using predator–prey models [42, 128]. Execution efficiencies have also been used as a benchmark to show the modelling ability of Repast Simphony [148] and to compare three approaches to simulation modelling: System Dynamics, Discrete Events and ABM [16].

The Sugarscape model is an artificial society model presented by Epstein and Axtell in their book *Growing Artificial Societies: Social Science from the Bottom Up* [40]. This model was replicated by several ABM platforms, such as NetLogo[5], MASON [14] and Repast [130]. Agents in the basic Sugarscape model follow very simple rules. They move towards deserted areas with high levels of sugar resources. The Sugarscape Wealth Distribution model, as described by Epstein and Axtell, is complex in terms of the relations between agents.

Boids or flocking models (introduced in section 3.3.1) have also been widely used to measure the modelling ability of some ABM platforms [49, 117, 128, 101].

[5]http://ccl.northwestern.edu/netlogo/models/community/Sugarscape

Rousset et al. [133] used their reference model [132] to benchmark 10 existing platforms that support parallel and distributed systems. Their model is based on three main behaviours for each agent: 1) agent perception, 2) agent communication and 3) agent mobility. This benchmark model is used to evaluate the ability of each platform regarding their parallelism support. A large and growing body of literature has focused on the comparison between parallel and serial execution methods to run simulations [42, 83, 36, 1, 34, 123].
All ABMs reviewed above were used as benchmarks for two purposes: to evaluate modelling capabilities of platforms and/or to make comparisons between simulators. Observing system performance and how applications behave as the size of the agent population increases is the main focus of benchmarking in the majority of these studies. System scalability is not the only issue that may affect the overall performance of an ABM simulation; there are some issues that need to be dealt with to create a standard benchmark model that meets all ABM criteria.

5.3 The benchmark Model

Our model is based on the concept of an abstract molecular chemical system in which a particle-based simulation represents each molecule in the system as an individual entity. This entity has attributes, such as position, velocity and molecule type. Entity movements and the reactions within the system will be computed using these attributes to update system behaviour. The movement of the molecule (agent) will follow Brownian Dynamics methods [155], where each agent is represented as a point-like particle moving randomly in the environment.

This type of model is relevant to a wider class of ABMs. For example, both cellular and social system models have similar behaviours when considered from the viewpoint of mobile agents with local interactions, birth and death and binding (combining). To make this model controllable and variable with respect to homogeneity and complexity, we propose a parameterised reaction-diffusion model with different rules. Our model is able to convert formula syntax (such as `A+B=C`) that represents a chemical reaction to a model specification with a number of mobile agents that can communicate with each other while moving within a virtual environment. Any combination of letters can be used by a

number of equations to vary the complexity and homogeneity of the population.

A simple reaction will occur when one A molecule combines with one B molecule to produce a C molecule, assuming that `A+B=C` represents the relationship between the three molecules. The model that results from the example above contains three agents, as follows: agent A (master agent), agent B (slave agent) and agent C (combined agent)[6]. Each of these agent specifications is defined by a set of variables and functions that help to establish the simulation. At the beginning of the simulation, two populations A and B randomly move within the continuous environment (Brownian motion). Individuals from each population aim to interact with an individual from the other population in a mutually exclusive pairwise interaction (i.e. one A will interact with one B at a closest pre-calculated point). This model also consists of an additional probabilistic test to increase stochasticity on agents' movement through two variables. The interaction rate is controlled by maximum individual movement and the interaction radius as model parameters. Agent B will send its location, and then agent A will choose the closest B, which will reply with its ID. Once the ID of B is confirmed (a requirement of performing paralyse operations in parallel), both agents will die and produce the new agent C. To ensure reliable agent populations for benchmarking and evaluation, once a pairwise interaction occurs, there is a counter for each population type to update the population size of each type.

5.3.1 Implementation

This section consists of three parts: 1) how the benchmark model is implemented using FLAME GPU, 2) the state diagram of the model and 3) dynamic model generation from simple formula syntax. The FLAME GPU implementation of the above-mentioned example consists of three agents A, B and C. Each agent is defined by a set of variables, transition functions, start and end states and communication messages, as shown in Table 5.1.

At the beginning of the simulation, agents A and B are moving randomly using their move functions to update their locations during each cycle, as shown in Fig 5.1 Part A. Agent B will use `send_locationB` to output a locationB

[6]It is named this way to differentiate its functions within the simulation, and it is a convenient way to construct our model. Every equation consists of one master agent, one combined agent and one or more slave agents.

Table 5.1: Agent specifications

Agent Type	Memory	Function Name	Function Description
Master agent	Agent ID Agent Position: X,Y,Z Closest_id Closest_point state	1.move_A 2.need_locationB 3. send_bindB 4. created_C 5. death_A	1.To update A's location 2.Choose closest B 3.Send request to closest B 4.Output agent C 5.Remove agent A from simulation
Slave agent	Agent ID Agent position: X,Y,Z Closest_id Closest_point state	1.move_B 2.send_locationB 3. receive_bindB 4.send_combinedB	1.To update B's location 2.Send B location 3.Verify and choose closest A that is ready to bind. 4.Send notification to A to combine and then remove agent B from the simulation
Combined agent	Agent ID Agent position: X,Y,Z Closest_id Closest_point state	move_C	To update C's location

message holding all B information (agent ID, location, etc). After that, agent A will get all B's locations using a need_locationB function that inputs the locationB message. This function will calculate the distance between A and B and then compare it with the binding radius. If the distance is less than or equal to the binding radius, the internal memory of A will be updated (the state variable will be set equal to 2, the defined value of binding (2 is the defined value of the combined state), and the closest ID of an A agent and the closest point will be stored). The send_bindB function will output bindB messages holding the updated information for agent A (only messages that have the state variable equal to 2 as a function condition (*An agent function condition indicates that the agent function should only be applied to agents which meet the defined condition)*, which are in the correct state specified by current the

current state [122]). In the next step, the `receive_bindB` function will input bindB messages to check for the closest A that is ready to combine. B's internal memory will be updated (the state variable will be set to 3 (where 3 is the defined value of the dead state), and the closest ID and closest point will be stored) after finding the closest A that is ready to combine. The `send_combinedB` function will output combinedB messages that meet the condition (the state variable is equal to 2), and the B agent will be removed from the simulation. The next function will be `created_C`. This function will input combinedB messages (only messages that meet the condition that the state is equal to 3), output agent C and update A's internal memory (the state variable will be updated to meet the next function condition). All A's that meet the condition of `death_A` will be removed at this stage. A visualisation of the model after a number of iterations is shown in Fig 5.1 Part B. Each time that one A and one B agent combine, they produced one C agent, as shown in Fig 5.2.

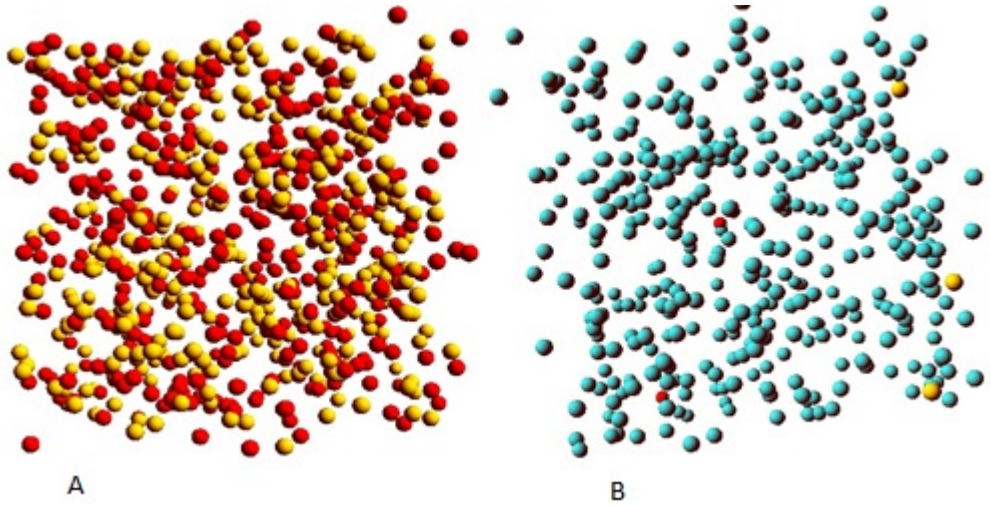

Figure 5.1: Part A: Screenshot of the first iteration showing agents A (red) and B (yellow) moving randomly. Part B: Screenshot after 100 iterations showing agents C (blue) moving randomly and two of A (red) and two of B (yellow) still moving.

5.3.2 The state diagram of the model

The representation of agents as a state machine is shown in Fig 5.3 for the A+B=C example. This diagram represent a single iteration of the simulation, where each type of agent will move from the starting state to the end state, completing each function in turn. The diagram is divided into nine layers, showing the agent-transition functions and the communication dependency messages

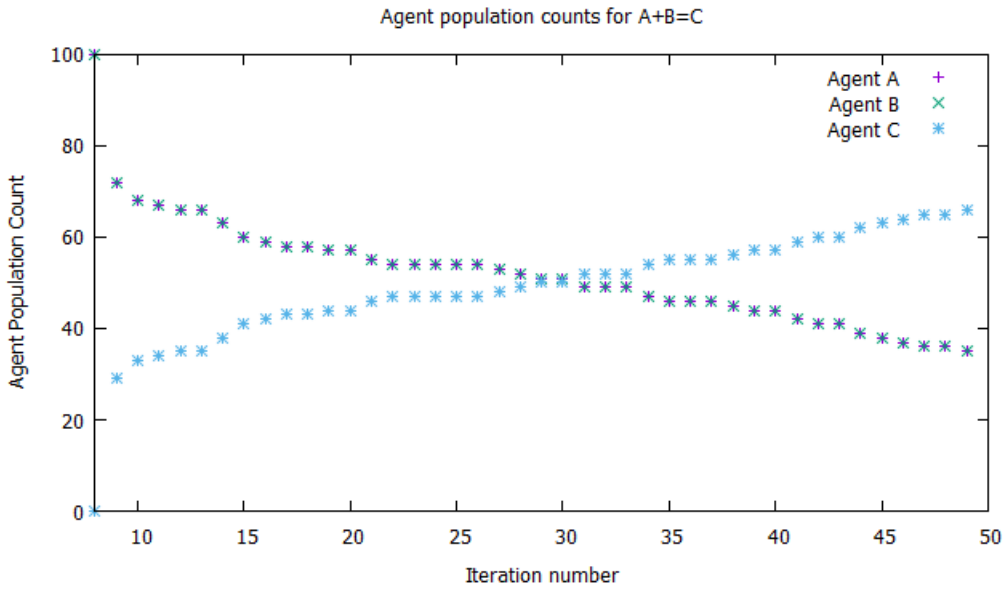

Figure 5.2: Histogram generated by the model during the run time of the simulation presenting the agent population count for A+B=C against iteration number. This is to indicats that the implementation of the model behaviour was correct.

(green) for each agent. Both agents A and B perform their movement behaviour (move function) in the first layer, where agents use Brownian motion. B agents in the second layer execute the `Send_locationB function`, broadcasting their location within the simulation environment to the Location message list. This message list is iterated by A agents in the `Need_locationB` function, where A agents select the B agent they wish to interact with. In the fourth layer, the `Send_bindB` function broadcasts all selected A agents ready to combine into the `BindB` message list. In the fifth layer, B agents iterate the message list, deciding which A interaction they will participate in. The confirmation message will be sent to the `CombinedB` message list using `Send_combinedB`, and dead B agents will be removed from the simulation. In the seventh layer, A agents execute the `Created_C` function, and B agents iterate the confirmation messages to outputs of the new C agents. `Death_A` function is responsible for removing dead A agents from the simulation. In the last layer, C agents move by applying the (`Move_c`)function.

5.3.3 Model Generator

In the previous section, a simple reaction of A+B=C was implemented using FLAME GPU. Duplicating the behaviour while adding a different type of

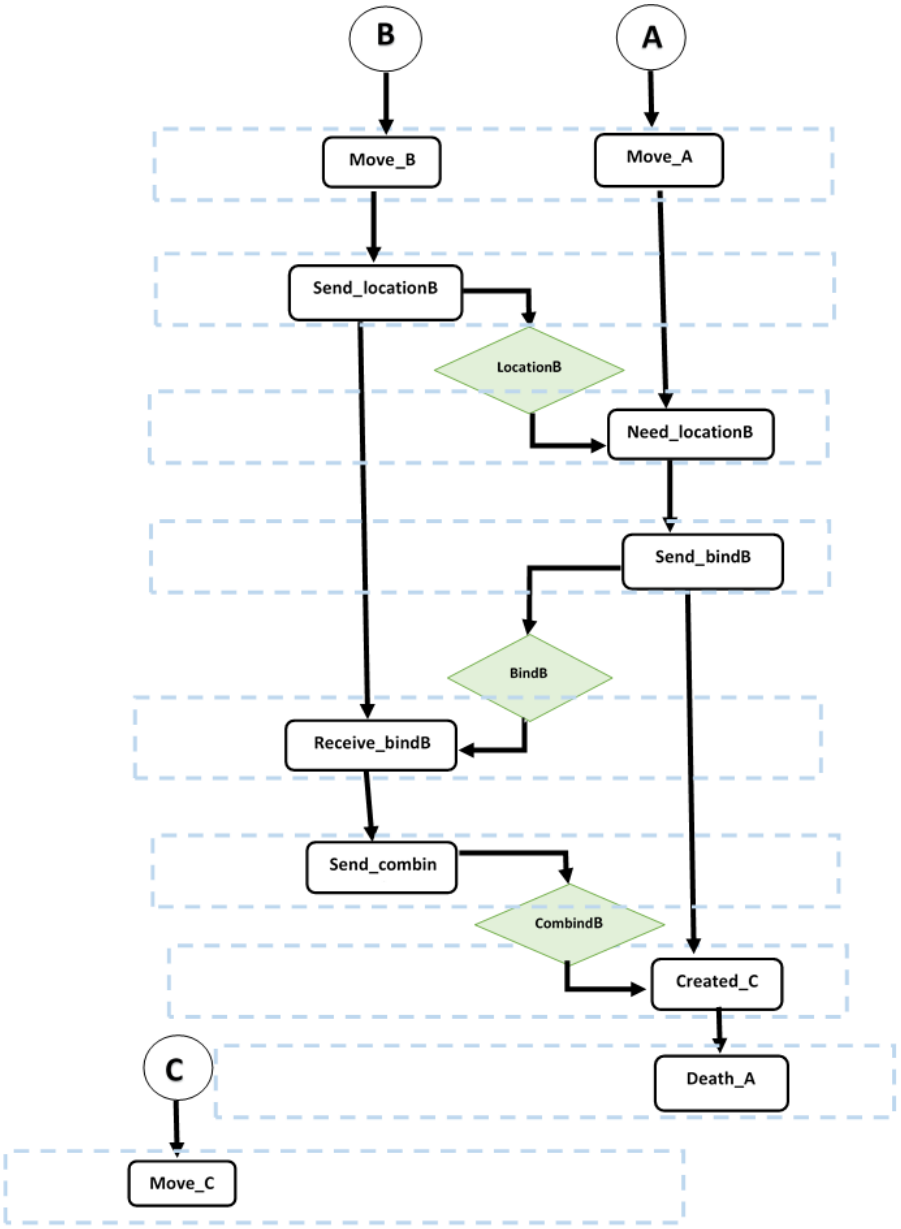

Figure 5.3: State graph of the model that represents A+B=C. The diagram illustrates the order of agent functions (black rectangles) and the interactions between individual agents using message lists (green rhombuses) through a single iteration. Functions can run simultaneously within a layer, indicated by blue dashed boxes.

molecule to the same simulation will result in more divergence, which is likely to effect simulation performance, particularly on GPUs for which [22] highlighted divergence as a key performance criterion. To save time and effort, and to implement several chemical reactions at the same time automatically, a model generator is required.

This section presents a FLAME GPU model generator that can easily convert formula syntaxes to a valid model specification. This generator, after parsing the formula syntax, will output three files that are required to run a FLAME

GPU model: 1) a FLAME GPU XML model (XMLModelFile.xml) file 3.3.2 that consists of model specifications, 2) a function.c file 3.3.3 that holds the scripted agent functions and 3) initial values of each agent for the simulation state data, which are stored in a FLAME GPU XML file (0.xml).

The representation of a state machine of the generated model that presents (A+B+C=D) is shown in Fig 5.4, and the histogram of agent counts during simulation of the same model is shown in Fig 5.5. Both figures show the results that we expected in terms of how the model (A+B+C=D) should behave. The extra slave agent was easily added to the model, resulting in more divergence of agent behaviour within a population and more divergence of behaviour within agents (the extra transition functions that have been added are shown in Fig 5.4). There is extra depth in the state diagram compared with Fig5.3, as the divergence focuses on one type of agent (master agent) when more slave agents have been added to the same equation.

The model generator also has the ability to accept more than one equation at the same time (e.g. example `A+B=C` and `D+E=F`) under some conditions, such as (must be written in separated lines and the letters must not be the same in the same model to avoid the confusion). Converting more than one equation to movement agents in FLAME GPU increases the width of the state diagram every time an extra equation has been added. Thus, the main idea of the model generator is to duplicate the main behaviour of a simple model and produce a huge model that contains a large number of agents with different types and complex relationships.

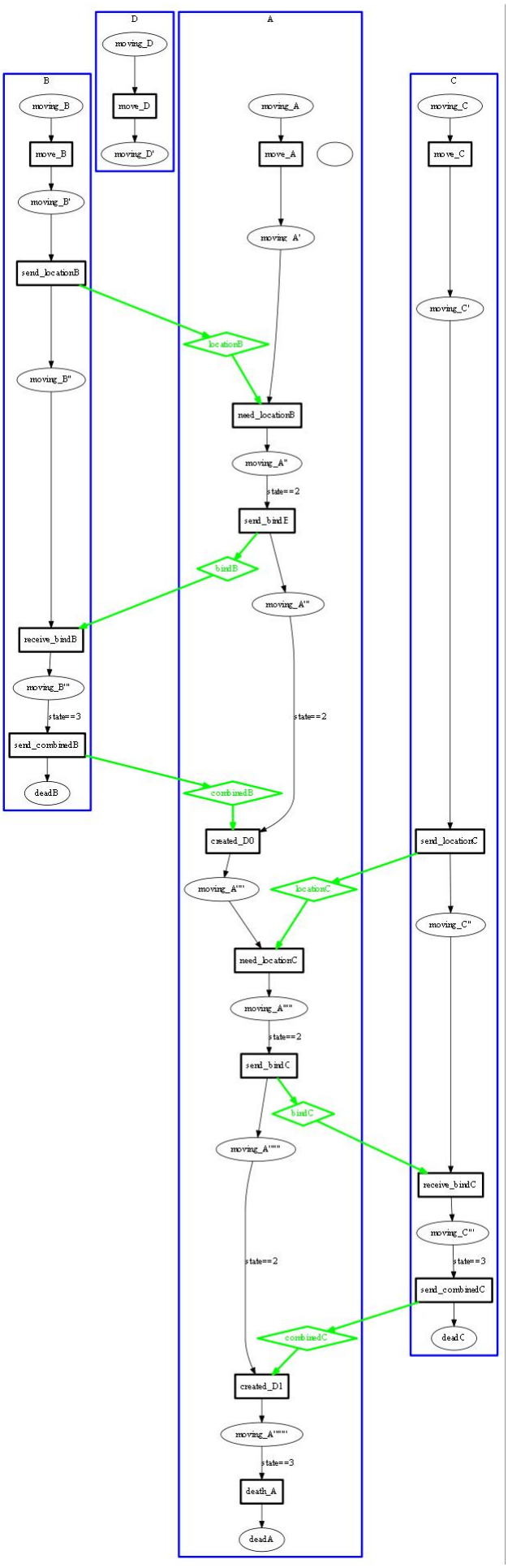

Figure 5.4: State graph of the model that represents A+B+C=D.

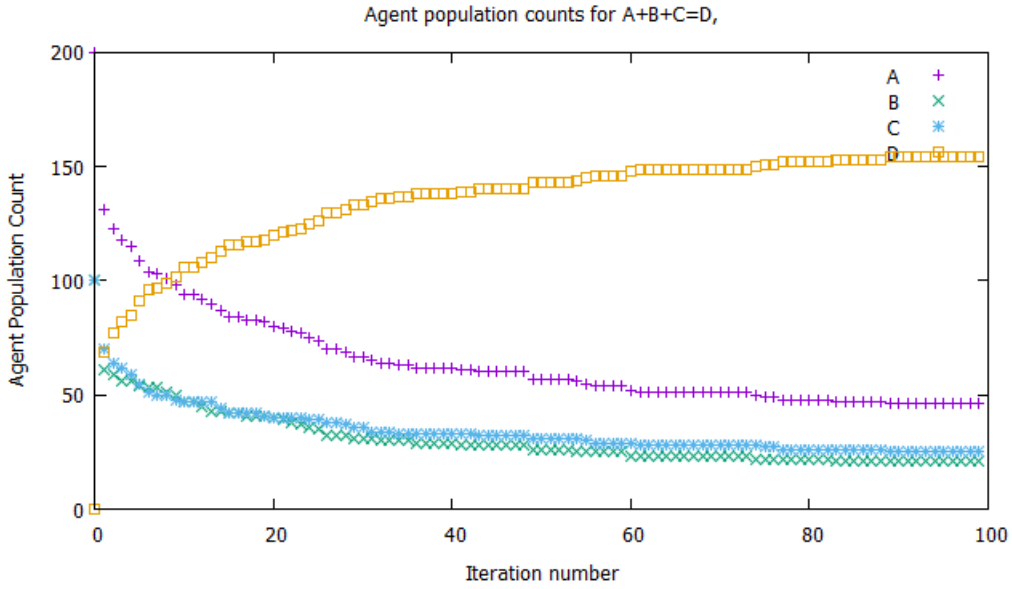

Figure 5.5: Histogram generated by the model during the run time of the simulation presenting the agent population count for A+B+C=D against the iteration number. This histogram shows that the number of the master agents (A) is decreasing, the number of combined agents (D) increases by the same amount and the total number of all slave agent (B,C) types is equal to the number of master agents in each cycle.

5.4 Benchmarking Results

The model generator can be used to vary the model in different ways and allows modelling of different types of chemical reactions. The machine used for benchmarking all three experiments has an NVIDIA TITAN Xp graphics card with 5120 CUDA cores and 12 GB of memory. Each experiment was run 30 times for each sample, and the median value of the execution time was calculated and presented in a graph for each experiment.

5.4.1 Divergence within a population:

The purpose of this benchmark is to observe the FLAME GPU performance when varying the type of agents within the population. This benchmark starts with a simple model with three types of agent, ten agent functions and three type of messages and ends with 30 agent types, 300 agent functions and 30 message types. Adding more equation input lines (every line contains three different types of agent) increases the execution time, as shown in Fig 5.6, and the interquartile range values can be seen in Fig 5.7. This benchmark was implemented using an agent population of 100000 for each type of agent with the same environment size, and each simulation was performed for 100 iterations.

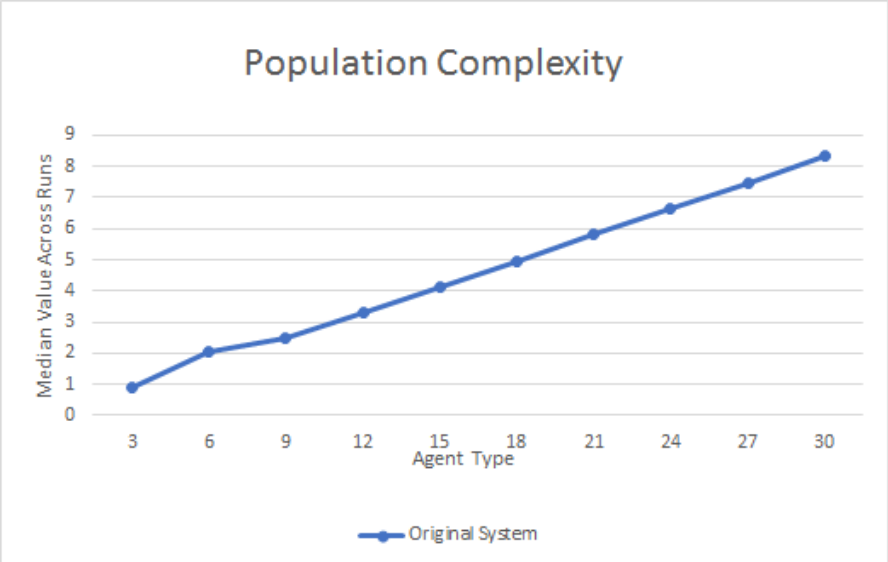

Figure 5.6: Median value of the execution time of the same environment size against the type of agent that has been added at every step

Agent Type	3	6	9	12	15	18	21	24	27	30
Median	0.87125	2.033481	2.483887	3.287727	4.119741	4.962644	5.809515	6.651118	7.486232	8.327183
Q1	0.869166	2.024799	2.491427	3.280865	4.112702	4.958915	5.820551	6.651966	7.489308	8.306199
Q3	0.882507	2.038594	2.508702	3.29408	4.122689	4.975486	5.841894	6.668459	7.508485	8.329633
Min	0.856331	2.00188	2.471746	3.269925	4.097579	4.947163	5.808396	6.638768	7.480114	8.291731
Max	0.888631	2.047367	2.512853	3.304941	4.137091	4.996626	5.862757	6.688544	7.51509	8.35633
Range	0.0323	0.045487	0.041106	0.035016	0.039513	0.049463	0.05436	0.049776	0.034976	0.064599

Figure 5.7: Interquartile range values in seconds for simulation runs (Agent complexity benchmark)

5.4.2 Divergence within an agent:

This benchmark gives us the median value of the execution time for increasing slave agent types (more chemicals per line). This experiment will increase divergence within the master agent of this line. Adding a new chemical will extend the master agent functions, which means more layers within the function layers. In FLAME GPU, function layers represent the control flow of simulation processes [122]. All agent functions are executed in a sequential order to complete one iteration, and by adding more functions for the same agent, the total number of layers will increase. More layers indicates serialised execution and low utilisation of the device. This can be observed in the results in Fig 5.8, and the interquartile range values can be seen in Fig 5.9. This benchmark was implemented using an agent population of 100000 for each type of agent with the same environment size.

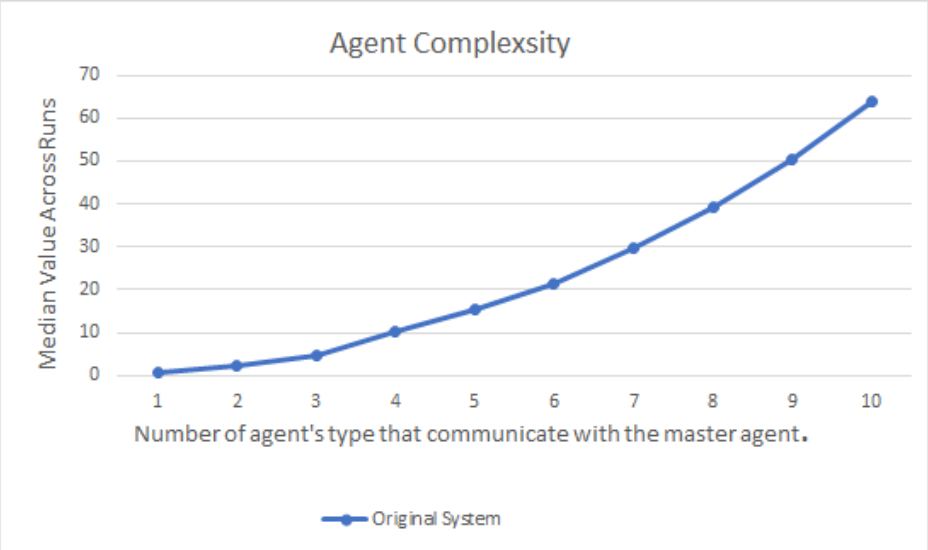

Figure 5.8: Median value of the execution time against the number of slave agents that have been added every time

No Slaves	1	2	3	4	5	6	7	8	9	10
Median	0.87125	2.250896	4.854636	10.2274	15.30253	21.56345	29.84382	39.29331	50.51696	63.68922
Q1	0.870217	2.23803	4.847447	10.21194	15.27613	21.55757	29.83693	39.29883	50.43648	63.69116
Q3	0.882507	2.260658	4.865583	10.25211	15.30724	21.59243	29.88592	39.39018	50.52452	63.79817
Min	0.857556	2.221129	4.839978	10.18812	15.24989	21.51234	29.71215	39.27788	50.37611	63.67521
Max	0.888631	2.286164	4.874989	10.28189	15.32472	21.63109	30.0171	39.44478	50.6571	63.97801
Range	0.031075	0.065034	0.035011	0.093764	0.074827	0.11875	0.304946	0.166898	0.280989	0.302806

Figure 5.9: Interquartile range values in seconds for simulation runs (Population complexity benchmark)

5.4.3 Population sizes:

The goal of this benchmark is to measure the ability of ABM systems to scale. The population size of each agent type starts with 100000 agents and ends with 800000 agents. This benchmark uses A+B=C as an example to run this experiment for 100 iterations each time. The performance of implementing our model on FLAME GPU with respect to agent population size is shown in Fig 5.10, and the interquartile range values can be seen in Fig 5.11.

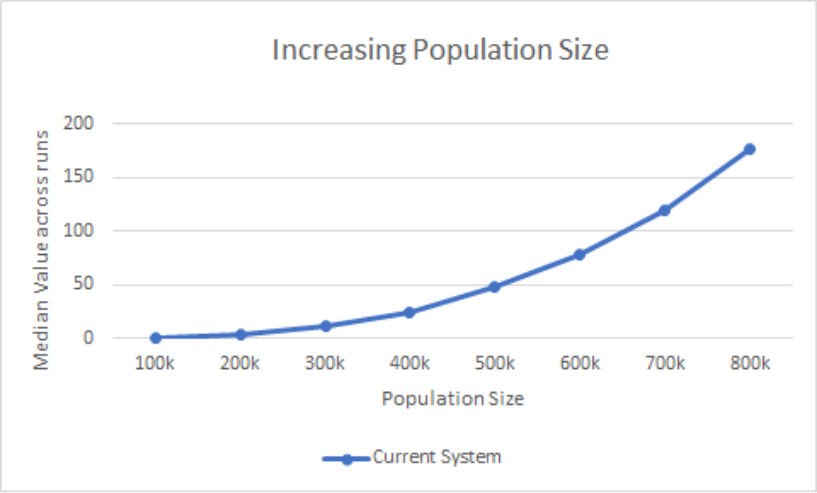

Figure 5.10: Increasing population size led to increased simulation execution time.

Agents	100K	200k	300k	400k	500k	600k	700k	800k
Median	0.87125	3.87119	11.73987	24.43963	47.95274	78.54813	119.9293	175.99
Q1	0.869166	3.856018	11.73954	24.27348	48.01495	78.30632	119.9422	175.5177
Q3	0.882507	3.877342	11.82492	24.5759	48.12952	78.60423	120.4133	176.0508
Min	0.857556	3.839989	11.72143	24.12413	47.91798	78.1227	119.2942	175.1189
Max	0.888631	3.908828	11.84545	24.62934	48.2668	79.0993	120.795	176.5279
Range	0.031075	0.068838	0.124012	0.505209	0.348828	0.976602	1.500797	1.408969

Figure 5.11: Interquartile range values in seconds of simulation runs (Scalability benchmark)

5.4.4 Level of communication and complexity:

Two changes have been made to agent behaviour to slow down the simulation and vary arithmetic intensity within agent functions: 1) varying the interaction radius and 2) decreasing the agent movement speed. Fig 5.12 shows the relationship between decreasing the interaction radius with increasing overall processing time to produce 50 agents C from the A+B=C equation with the same movement speed. This experiment allows agents to move for a longer period of time until reaching the needed radius. During this movement, several operations occur, such as calculating agent position and sending and receiving messages between agents looking for the nearest agent to combine with. The next experiment is shown in Fig 5.13, which shows the relationship between slowing down the agent speed and the number of iterations required to produce 50 agents. This experiment has been implemented with a constant radius and same environment size. Slowing down the movement speed allows additional operations during the simulation, which helps to measure the ability of the system to handle many computational operations for a long time and how to utilise the resources.

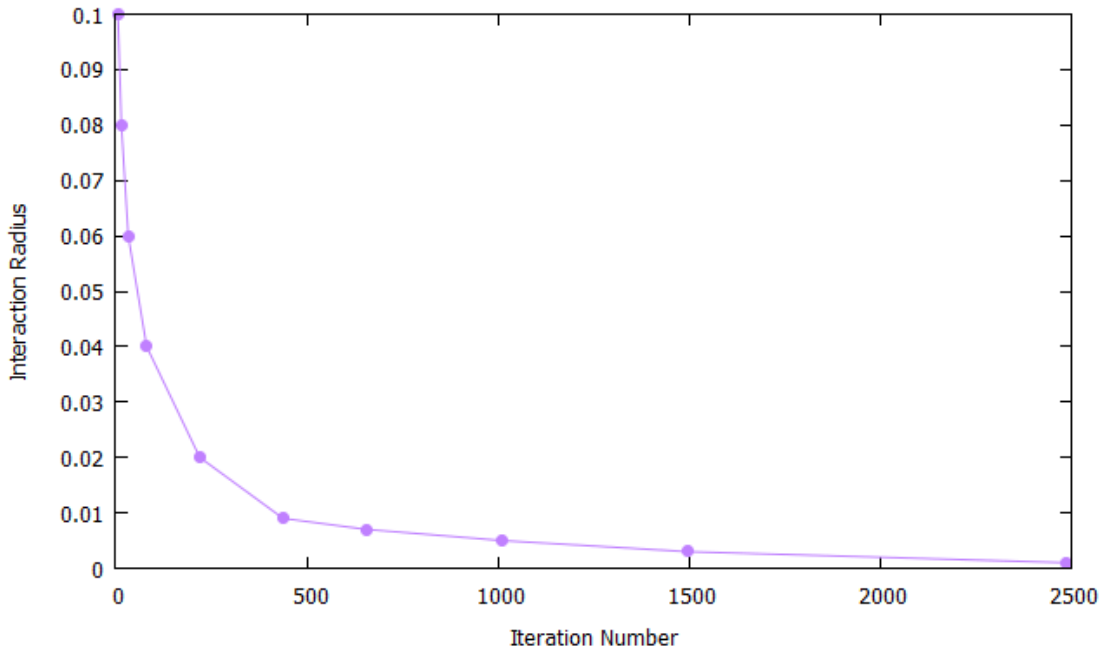

Figure 5.12: Decreasing the interaction radius led to increased time to produce 50 agents

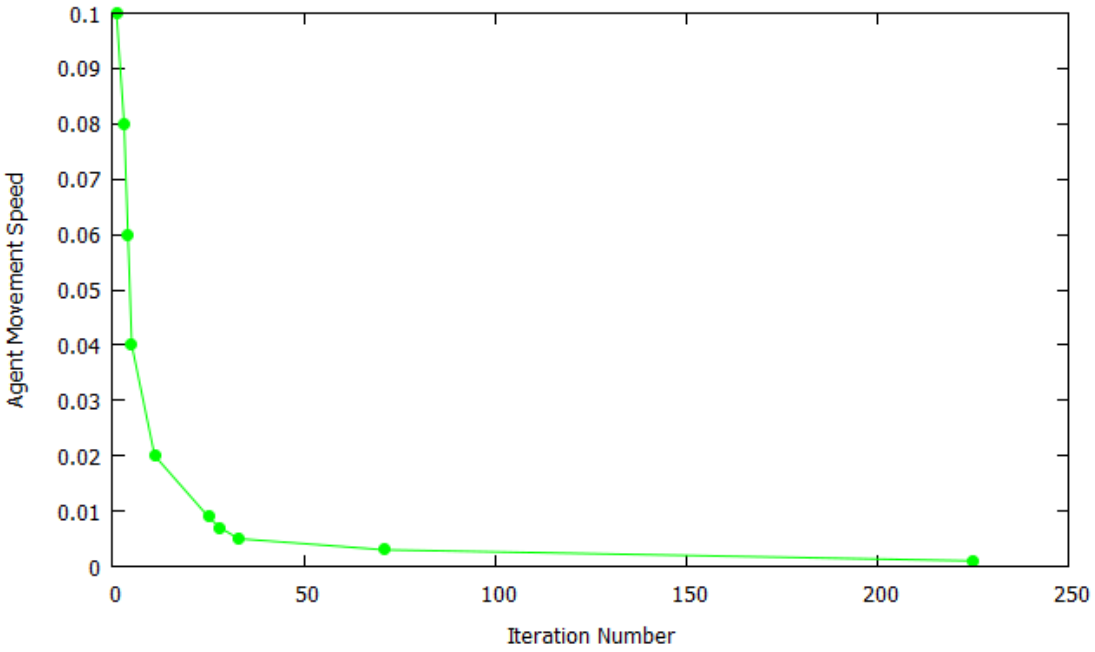

Figure 5.13: Decreasing the agent movement speed led to increased time to produce 50 agents

5.5 Summary

Four benchmark experiments have been carried out, demonstrating the ability of this benchmark model to examine each element. The first two experiments focused on increasing agent and population divergence, which led to an increase in execution time due to the additional agent functions, messages and communication information that is held by these messages. The third experiment showed that we could easily scale the population size of this model to measure the system scalability. The results showed that scaling the population size led to varying the execution time from 0.9 seconds per 100 iterations for 100000 agents to 176 seconds per 100 iterations for 800000 agents. These results were based on the median value of the execution time. In the last experiment, computational complexity was varied by decreasing the value of two variables that are used within agent functions to updating agents behaviour. This experiment caused the model to reach a steady state at a slower rate, allowing control of the arithmetic intensity of agents within the model.

Divergence in the agent population increases homogeneity within the model and results in lower utilisation of the GPU, particularly when executing agent functions, as confirmed by fewer agents executing the same function (additional serialised of execution) and our benchmark model. The obtained results will be used for assessing simulator improvements to achieve improved scaling

with respect to population-level divergence and better overall performance to increase the population size. The performance results indicate that our benchmark model is suitable for use as an experimental tool to evaluate the modelling capabilities of an ABM system if it is replicated in a suitable way.

Chapter 6

The Impact of Combining Agents Functions on Overall Performance

6.1 Introduction

The second chapter highlighted that data dependency is one of the critical issues in real-time HPC applications. Adopting an approach that reduces memory access time could minimise this problem. Conceptually, the agents in a simulation could execute in any random order; however, this is very inefficient, as agents are waiting on results from each other. In FLAME GPU, the parallel execution of agents is deliberately maximised by ordering all processing based on perceived dependencies (through messages). Agent functions are organised in layers (explained in section 3.3.2.5), where all functions within a layer execute in parallel, and agent functions placed in subsequent layers execute after those in preceding layers. This ensures that functions can execute without any blocking or race conditions. However, an optimising process could be devised to identify message and data dependencies between agents automatically, and this could be used to re-organise the granularity of agent functions, splitting and merging them in order to maximise device utilisation and minimise data movement. This would reduce the execution time of the simulation by increasing the amount of useful parallelism.

This chapter presents a functional approach for splitting and merging func-

tions based on function dependency analysis. As mentioned earlier, this approach is specific to the FLAME GPU, and it is an attempt to discover a way to take advantage of the data dependency between FLAME GPU functions to reduce simulation execution time. An investigation into data dependency between agent functions will be applied to the benchmark model presented in the previous chapter. Section 6.2 discusses data dependency between FLAME GPU model functions and their effect on performance. Section 6.3 presents an experiment for discovering dependencies between model functions. Section 6.4 describes the implementation of a functional approach to minimise the number of model functions. In section6.5, strengths and limitations of this approach are presented. Finally, section 6.5 reviews the results of applying this approach to the benchmark model and compares it with the original model.

6.2 Dependencies Between Model Functions and their Effect on Performance

The implementation of the original FLAME uses dependencies to determine function and communication order, as described in section 3.3.2.5. Listing 6.1 shows the syntax for organising functions into layers. The pair function1, function2 are placed in one layer, as they are presumed to act independently with no shared message inputs or outputs. The third function3 is placed in a subsequent layer and thus may depend, for its inputs, on the outputs of functions in the previous layer.

An iteration is a full cycle in which all layers are executed at most one time; however, some functions in each layer may not be triggered if the state conditions are not favourable. Even so, in most cases, all functions of the same agent are executed in a sequential order to complete one iteration. If our goal is to minimise the time taken to execute one complete iteration, then we should seek to minimise the number of layers in one iteration since layers are executed sequentially, and each function require data movement of an agent's internal memory. To do this, we should seek to pack more agent functions into fewer layers. This can only be done if we can identify the real data dependencies between the functions (as opposed to perceived dependencies).

For example, if two functions of the same agent have been placed, for conceptual modelling reasons, in subsequent layers and it is found that there is no

external data dependency forcing this separation, then they can be combined in a single function. Similarly, agent functions that have multiple data dependencies may be split into simpler functions with simple data dependencies and may possibly be re-grouped into larger functions according to their dependency on the same data.

In this way, it may be possible (depending on the agent model) to reduce the global number of agent functions by a process of splitting and merging, and for these functions to be packed into fewer layers. A second consequence of merging functions that depend on the same data is that our later technique (see chapter 7) of accessing subsets of agent memory variables may also be used to minimise data movement. Section 6.4 describes an experiment merging two functions of the same agent. The next section presents the benchmark model dependency analysis with a simple example.

```
<layers>
  <layer>
    <gpu:layerFunction>
      <name>function1</name>
    </gpu:layerFunction>
    <gpu:layerFunction>
      <name>function2</name>
    </gpu:layerFunction>
  </layer>
  <layer>
    <gpu:layerFunction>
      <name>function3</name>
    </gpu:layerFunction>
  </layer>
</layers>
```

Listing 6.1: function layers

6.3 Dependency Discovery (Manual version)

Initially, the data dependencies between agent functions have been discovered manually for the benchmark model 5. As described earlier, this model is able to convert formula syntax that represents a chemical reaction to a large number of movement agents. These agents can communicate with each other and capture important ABM characteristics. This experiment focuses on a model that represents the simple equation A+B=C . This model consists of three types of agents and ten agent functions executed in nine layers. By analysing the agent functions file (available in Appendix A), we can track variable read and write operations within functions so that the dependency of later functions is based on earlier function writes. Fig 6.1 shows the state diagram of the model after manually evaluating all the data and message dependencies. As shown in the figure (marked with a red circle), there is no dependency between some functions. This can be noted in table 6.1; all the rows appearing in grey consists of functions with no data or message dependencies. This table summarises the dependencies of model functions. It lists each function's name, the owning agent and any other functions on which the function depends (conceptually). Each dependency is then analysed to see whether it constitutes an internal data dependency, an external message communication dependency or neither. Where there are no real data or message dependencies and the functions belong to the same agent, the two functions may be merged. This may result in reducing the number of layers, which is hypothesised to decrease iteration execution time. The next section discusses this functional approach in more detail.

Table 6.1: All agents functions of the model that represent A+B=C and their relation dependencies to other functions

Function	Agent	Dependent On	Dependent Data	Dependent Message
Move_A	A	_	_	_
Move_B	B	_	_	_
Send_locationB	B	Move_B	Yes	No
Need_locationB	A	Move_A Send_locationB	Yes No	No Yes
Send_bindB	A	Need_locationB	No	No
Send_locationB	B	Move_B	Yes	No
Receive_bindB	B	Send_bindB Send_locationB	No No	Yes No
Send_combinedB	B	Receive_bindB	No	No
Created_C	A	Send_combinedB Send_bindB	No No	Yes No
Death_A	A	Created_C	No	No
Move_C	C	Created_C	Yes	No

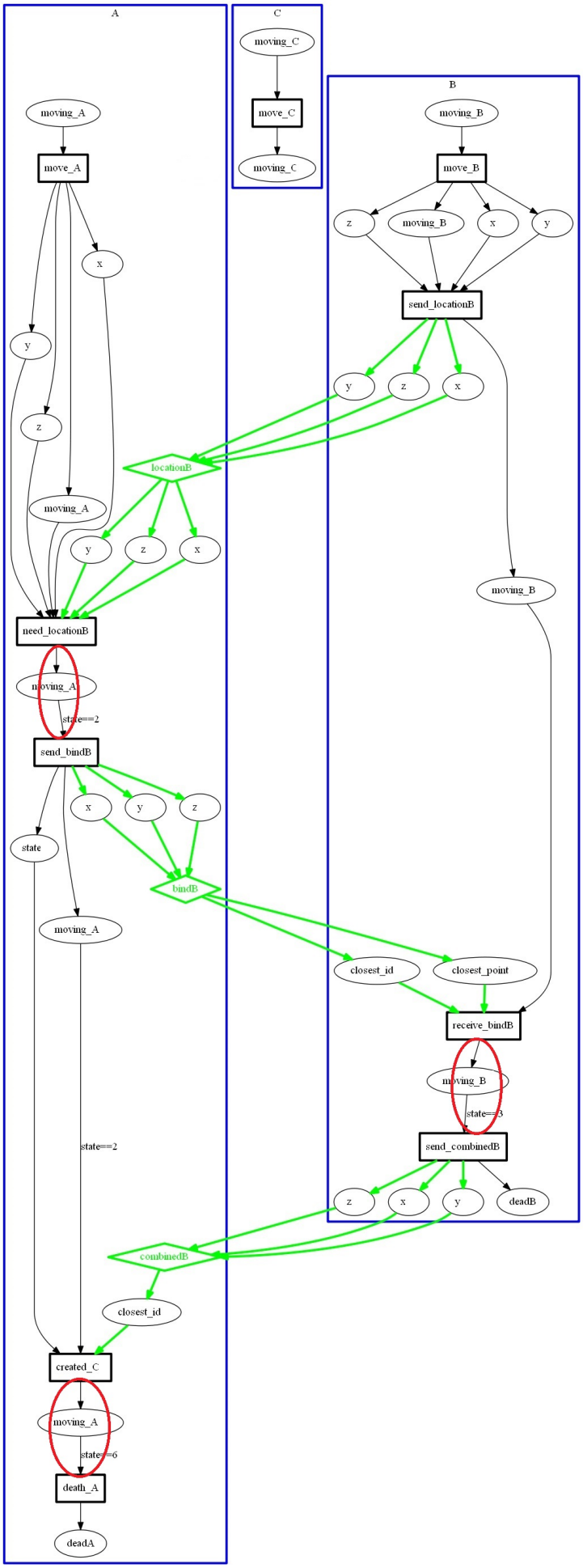

Figure 6.1: The state graph showing the data and message dependency and the possibility to combine some functions marked with a red circle.

6.4 Merging Functions That Have no Dependency

According to the results of the previous section, the manual dependency analysis of the benchmark model showed three independent functions, two of which belong to agent A (`send_bindB` ,`death_A`), while one belongs to agent B (`send_combinedB`). Each of these functions was combined with the preceding function, as shown at the state graph in Fig 6.1. The new state graph of the modified model can be seen in Fig 6.2. The benchmark model now consists of seven functions executed in six layers. Fig 6.3 (old version) and Fig 6.4 (new version) also present screenshots of how the function layers are represented in the XML model files. Reducing the number of agent functions also has the consequence of minimising data movement (agent variables do not need to be transferred as frequently from main memory). Table 6.2 shows the reduction in data movement. This table presents values of optimising a simple model that represents an A+B=C equation, as mentioned earlier. In order to evaluate the benefit of this manual optimisation (before we proceeded to automate this process in the future work), we wanted to generate scalable models of the benchmark particle simulation both with and without the functional approach. To generate the optimised models, a modification was made to the model generator to force it to output models in which splitting and merging are applied.

Table 6.2: The reduction in data movement achieved after merging some agent's functions in the benchmark model.

Model	No of Layer	No of Functions	Input	Output
The original model	9	10	120	120
The modified model	6	7	84	84

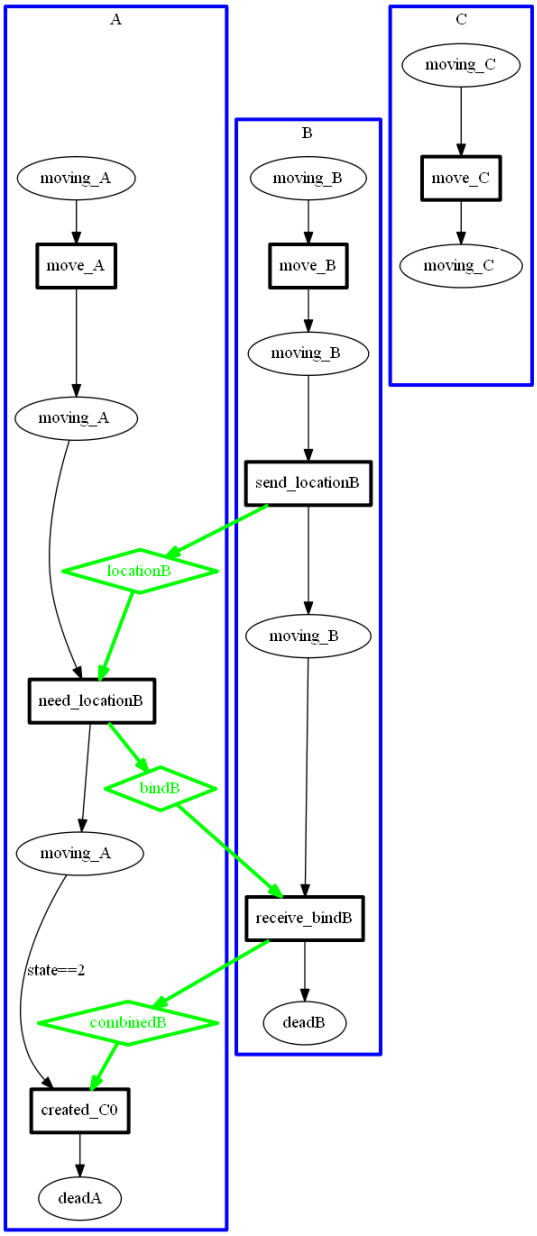

Figure 6.2: The state graph of the modified model

```
<layers>
    <layer>
        <gpu:layerFunction>
            <name>move_A</name>
        </gpu:layerFunction>
        <gpu:layerFunction>
            <name>move_B</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>send_locationB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>need_locationB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>send_bindB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>receive_bindB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>send_combinedB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>created_C0</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>death_A</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>move_C</name>
        </gpu:layerFunction>
    </layer>
```

Figure 6.3: Function layers of our benchmark model before merging (9 layers in total).

```
<layers>
    <layer>
        <gpu:layerFunction>
            <name>move_A</name>
        </gpu:layerFunction>
        <gpu:layerFunction>
            <name>move_B</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>send_locationB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>need_locationB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>receive_bindB</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>created_C0</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>move_C</name>
        </gpu:layerFunction>
    </layer>
</layers>
```

Figure 6.4: Function layers of our benchmark model after merging 3 functions (6 layers in total .

6.5 Benchmark Results

To evaluate the benefits of using the functional approach, this section presents a comparison of the results regarding the execution time of running both versions of the benchmark model using FLAME GPU. As described in Chapter 5, this model is based on the concept of particle-based simulation and accepts input parameters that control both system scalability and agent homogeneity. For system scalability, the population size for each agent type will be increased. In agent homogeneity, the focus will be on increasing the complexity for both individuals (by increasing communication) and the overall population (by increasing the diversity of agent type). Three different experiments have been performed to examine the performance efficiency of both versions: scalability, divergence within the population and divergence within an agent. The machine used for benchmarking both versions has an NVIDIA TITAN Xp graphics card with 5120 CUDA cores and 12 GB of memory. Each experiment was run 30 times for each sample, and the median value of the execution time was calculated and presented in a graph for each experiment.

6.5.1 Divergence within the Population

In this experiment, the number of agent types was increased in each step (by adding additional equations) to observe system behaviour while increasing complexity. This benchmark followed the same process that was applied to the original model in using the same environment size and the same population size, with the simulation running for 100 iterations. Fig 6.5 shows the processing time of both versions, with a noticeable improvement in performance, especially with a large model that consists of 10 lines of equations and 30 agent types. The interquartile range values cross-runs for the functional approach system can be seen in Fig 6.6.

6.5.2 Scalability

This benchmark will measure the scalability of the performance of both systems. The population size of each agent type starts with 100000 agents and ends with 800000 agents. This benchmark is based on the same example that was described in section 5.4.3 in Chapter 5. It is representative of scaling the model, and the simulation was performed for 100 iterations. A small improvement in

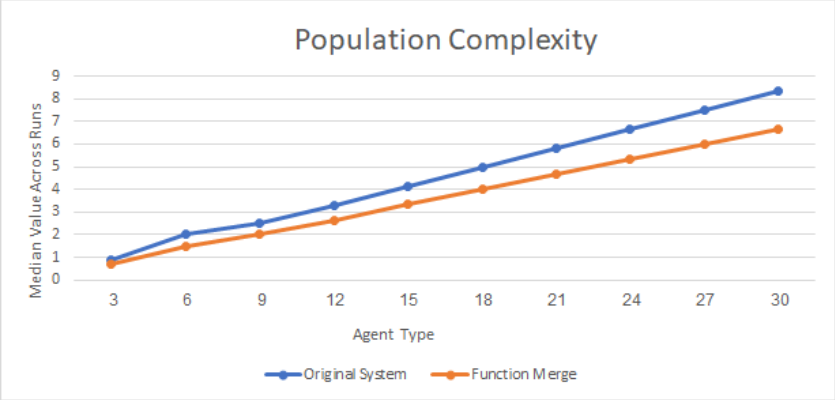

Figure 6.5: The difference in execution time between the original model (Red) and the modified model (blue) while increasing the divergence of the population.

Agents	3	6	9	12	15	18	21	24	27	30
Median	0.70963	1.475064	2.014931	2.644941	3.335814	4.010447	4.678957	5.325274	5.997972	6.670443
Q1	0.705863	1.411702	2.009625	2.641064	3.321613	4.004081	4.661425	5.316093	5.988794	6.667542
Q3	0.71311	1.506945	2.021232	2.65122	3.339962	4.013752	4.676665	5.331751	6.003925	6.679945
Min	0.699158	1.38938	1.997735	2.630969	3.311577	3.988072	4.644641	5.3005	5.969901	6.652541
Max	0.719868	1.695402	2.027996	2.662998	3.353513	4.031238	4.694167	5.344557	6.01395	6.693418
Range	0.02071	0.306023	0.03026	0.03203	0.041937	0.043166	0.049526	0.044058	0.044049	0.040877

Figure 6.6: Interquartile range values in seconds for simulation runs (Population complexity benchmark while applying the functional approach)

the execution time between both models can be seen in Fig 6.7, showing the median values for cross-simulation runs; the interquartile range values for the functional approach system can be seen in Fig 6.8.

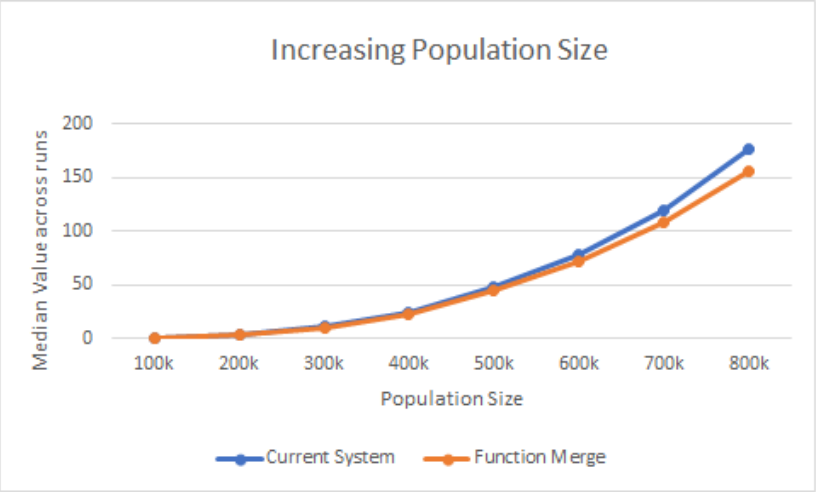

Figure 6.7: Comparison of median processing time values against population size, showing the original model (Red) and modified model (Green).

Population	100K	200k	300k	400k	500k	600k	700k	800k
Medain	0.70963	3.351862	10.5702	22.44416	44.71757	72.089	108.887	155.8051
Q1	0.705863	3.332767	10.554	22.42174	44.64993	72.03018	108.8083	155.6986
Q3	0.71311	3.362531	10.57981	22.47277	44.80305	72.11602	109.0339	155.9441
Min	0.699158	3.289884	10.50292	0.022447	44.50246	71.73711	108.6867	155.508
Max	0.719868	3.373552	10.60792	22.52085	44.86603	72.2711	109.317	156.2009
Range	0.02071	0.083668	0.105006	22.4984	0.363566	0.533992	0.630258	0.692922

Figure 6.8: Interquartile range values in seconds for simulation runs (while applying the functional approach and increasing the population size)

6.5.3 Divergence within the Agent

The same criteria for benchmarking the original model were used to run this benchmark on the modified model. This benchmark gives us the average execution time for increasing slave agent types. This experiment increases divergence within the master agent. Adding a new agent type (via more complex equation formulas with additional terms) to communicate with the master agent will extend its functions, which results in more functions in each layer every cycle. This benchmark was implemented using an agent population of 100000 for each type of agent with the same environment size, and each simulation was run for 100 iterations. The results of both benchmarks are shown in Fig 6.9. Figure 6.8 shows the interquartile range cross-run values for the functional approach system.

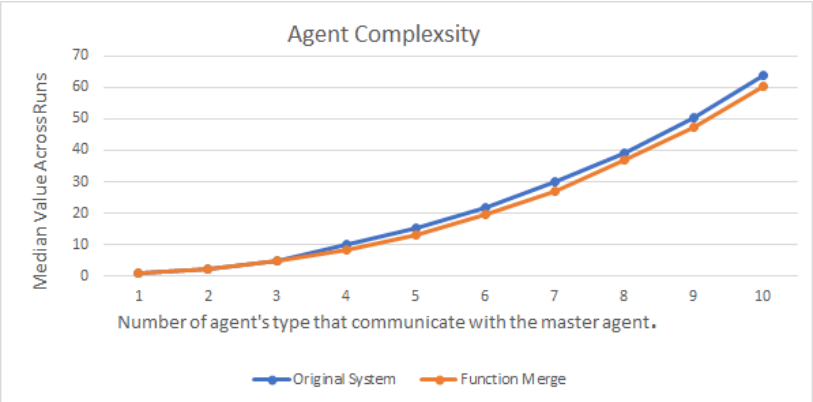

Figure 6.9: Comparison of median processing time values against the number of communicating agents (slave-to-master), showing the original model (purple) and modified model (green).

No Slaves	1	2	3	4	5	6	7	8	9	10
Median	0.713745	2.033481	4.642631	8.229303	13.28282	19.56345	27.13327	36.75978	47.51585	60.39218
Q1	0.705863	2.024799	4.625289	8.21073	13.24948	19.55757	27.14193	36.75149	47.43215	60.36077
Q3	0.71311	2.038594	4.651029	8.249311	13.29608	19.59243	27.18839	36.80522	47.51247	60.40311
Min	0.699158	2.00188	4.613515	8.182297	13.20861	19.51234	27.11796	36.72038	47.37493	60.30528
Max	0.719868	2.047367	4.679672	8.286701	13.32372	19.63109	27.21214	36.81333	47.60673	60.47154
Range	0.02071	0.045487	0.066157	0.104404	0.11511	0.11875	0.094178	0.092953	0.231801	0.166266

Figure 6.10: Interquartile range values in seconds for simulation runs (agent complexity benchmark while applying the functional approach)

6.6 Strengths and limitations

The benchmark results above showed that the functional approach has the ability to enhance the FLAME GPU performance by saving execution time in each simulation cycle by minimising the number of function layers. However, the main strengths of this approach were the motivation to automate the process of discovery of data dependency and the use the data to enhance the system performance by applying a data-aware approach (7.4). However, some limitations should be noted:

- This approach is specific to the FLAME GPU to reduce the execution time of the single iteration.
- This approach does not reduce the time required for memory access during the process of iterating message lists.
- This approach had a small direct effect on the performance of well-structured FLAME GPU models.

6.7 Summary

In FLAME GPU, agent functions are organised into layers based on when their state conditions are enabled. In this way, maximal parallelism may be ex ploited by deterministically scheduling all agent functions that may be executed together. This leads to a sequence of layers, denoting the order in which agent functions may be executed. Every function within a layer may be executed in parallel, and functions in a subsequent layer must wait for functions in the previous layer to terminate.

In the case of data dependency between states, subsequent transition functions must wait for the new version of the memory to be updated. This chapter presented a new method that helped to reduce the simulation execution time. The method is based on merging some agent functions after discovering data and message dependencies. The benchmark model that has been presented in chapter 5 was used to evaluate the benefits of using this method. The comparison results showed that this approach improves performance where functions can be merged. However, we could benefit more from discovering dependencies by accessing only the required data. Accessing a subset of data during the simulation would reduce the amount of data that need to be moved. FLAME GPU models could benefit more from this method due to the way of handling data movement within the FLAME GPU. When an agent is modified by a function, all of its variable values are transferred from shared memory to a local structure passed into the function. When the function has terminated, the variable values are copied back to the GPU's main memory. For this reason, prior knowledge of dependencies between agents functions may help to reduce the costs of data movement.

As the discovery of dependencies has been made manually in this chapter, the aim of the next chapter is to automate the process of identifying the data and message dependencies between agent functions.

Chapter 7

A Data Aware Model for Agent Representation

7.1 Introduction

In the previous chapter, the discovery of dependencies between agent functions was implemented manually by considering functional dependencies. Knowledge of data dependencies allowed some agent functions to be combined, which can reduce the execution time of each iteration. This experiment was applied to one model to evaluate the benefit of extracting data dependencies from existing models in FLAME GPU. Furthermore, it is hypothesised that dependency analysis at the variable level provides an additional opportunity for optimisation with data movement.

This chapter discusses the implementation of dependency parsing by using Flex and Bison tools as mentioned in section 4.3. In FLAME GPU, agent behaviour is specified through the use of an agent function, which uses a C-like syntax with prefixing macros and function arguments, dependent on the XMML function definition. The implementation of the parser,(for parsing and analysing agent functions) requires the context-free grammar (CFG) for C and the additional syntax for FLAME GPU's agent function scripting. Section 7.2 describe the implementation of the scanning process and Section 7.3 discuss the implementation of the parser.

This chapter also introduces a proposed modification to the X-machine model to explicitly model data dependencies. It focuses on how to use dependency information to optimise the simulation of FLAME GPU models. This goal can be achieved through two steps. First, updating the agent specifi-

cation file to include data dependencies of each function. Second, modifying the FLAME GPU templates to generate optimised GPU code which is data aware and as such is able to minimise data movement. Section 7.4 addresses an alternative representation of an agent in FLAME GPU that allows accessing required memory. Section 7.5 covers merging meta-data that is produced by the dependency generator with an agent specification file. In section 7.6, the implementation of modifying FLAME GPU templates to access a subset of message and agent memory variables. Finally, the summary of this chapter is presented in section 7.7.

7.2 Implementing a FLAME GPU Scanner

Section 4.5 has described the tools that will be used to implement the scanning stage for the dependency parsing process. This section discusses how the scanner is implemented by identifying rules that match the C code and additional tokens that are found in FLAME GPU agent functions. The main file in the lexical analysis stage in our system is `(Scanner.l)`. The Flex tool identifies this file by identifying the tokens and generates a C file `(lex.yy.c)` for the next step.

The main part of implementing this stage is writing the Flex rules. As agent functions are written in C like language with a special syntax for GPU implementation, the Flex rules must include regular expressions to recognise three kinds of syntax: standard C code, certain special macros, and certain special formal argument names that are used in FLAME GPU agent functions.

7.2.1 C Tokens

FLAME GPU contains a number of templates used to generate the dynamic simulation code. Some of these templates generate C (with a small amount of C++ features like templates) function prototypes. Therefore, the rules need to include a collection of C, C++ templates and the preprocessing directives, as they will be included within the `functions.c` file. For ANSI C grammar, we used the version that is based on the 2011 ISO C standard[1]. For preprocessing directives and C++ grammar, we used the C++ BNF Grammar that is available in [91] to cover all tokens missing from the previous source. The full version of the rules that match all possible tokens that are required for C and C++ syntax, including preprocessing and Pragma processing can be viewed in Appendix B.

7.2.2 Special Tokens for FLAME GPU Functions

Two types of tokens exist in FLAME GPU functions: prefix macro definitions and function arguments. All agent functions are first prefixed with the macro definition. These macros are listed in the table 7.1 below with a summary description for each one. The scanner recognises this type of token as a keyword.

Table 7.1: The scanner recognises macro definitions of all FLAME GPU functions as a keyword.

Syntax	Description
__FLAME_GPU_FUNC__	Macro definition for all FLAME GPU functions
__FLAME_GPU_INIT_FUNC__	Macro definition for initialisation functions
__FLAME_GPU_STEP_FUNC__	Macro definition for step functions
__FLAME_GPU_EXIT_FUNC__	Macro definition for exit functions

For some of FLAME GPU function arguments, the scanner needs a method to extract tokens as Flex reads the source code as a sequence of characters and recognises matching units to the rules. As an example, in the case of FLAME GPU function arguments, such as the argument to the function shown below in listing 7.1 , which is a pointer to an agent structure of type `xmachine_memory_myAgent` called `xmemory`, the scanner reads this pointer as one word, but in fact, it contains three components as follows: `xmachine_memory_` as a keyword, `myAgent` as an agent type identifier and * as an operator. To solve this problem, we designed a function to recognise large textual units that are not included within the predefined keywords. This function is called `check_type()` and it works as below in listing 4.2 . Another function called `Xmachine_specifier_checker()` has been designed to analyse only function's argument pointer or referred to as (xmachine_specifier). All possible FLAME GPU syntaxes and extracted tokens can be seen in table 7.2. All the keywords that appears in this table have been added to the rules sections within the scanner.l file as shown in the Fig 7.1 below.

```
1 1 __FLAME_GPU_FUNC__ int function(xmachine_memory_myAgent* xmemory)
2 2 {
3 3 xmemory->x = xmemory->x += 0.01f;
```

[1]The rule section of Flex specification of ANSI C grammar:http://www.quut.com/c/ANSI-C-grammar-l.html

```
4 xmemory->no_movements += 1;
5 return 0;
6 }
```

Listing 7.1: Analysing large textual units

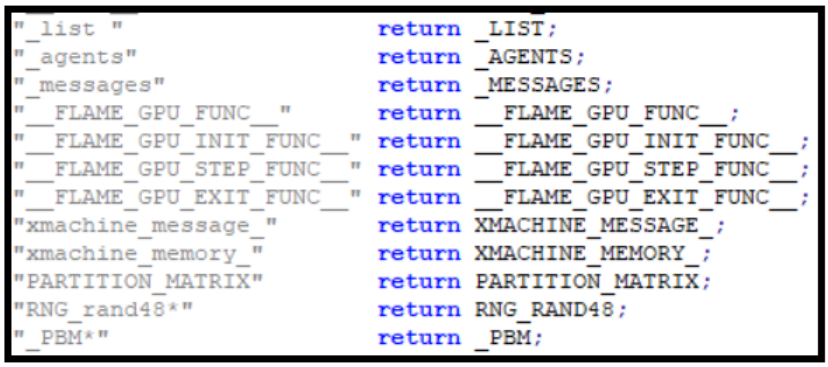

Figure 7.1: A part of the rules section from Scanner.l file showing FLAME GPU functions keywords

Table 7.2: FLAME GPU syntaxes and extracted tokens

FLAME GPU Syntax	Tokens	Token Type
xmachine_memory_AgentName*	xmachine_memory_ AgentName *	keyword Identifier Operator
xmachine_message_MessageName*	xmachine_message_ MessageName *	keyword Identifier Operator
AgentName_agents	AgentName _agents	Identifier keyword
MessageName_messages	MessageName _messages	Identifier keyword
xmachine_memory_AgentName_list*	xmachine_memory_ AgentName _list	keyword Identifier keyword Operator
xmachine_message_MessageName_list*	xmachine_message_ _list MessageName *	keyword keyword Identifier Operator
xmachine_message_MessageName_PBM*	xmachine_message_ MessageName _PBM *	keyword Identifier keyword Operator
get_first_MessageName_message	get_first_ MessageName _message	keyword Identifier keyword
get_next_MessageName_message	get_next_ MessageName _message	keyword Identifier keyword
partition_matrix*	partition_matrix *	keyword Operator
RNG_rand48*	RNG_rand48 *	keyword Operator

7.3 Implementing a FLAME GPU Parser

Section 4.6 in chapter 4 discussed the way of generating both syntax and semantic analysis using Bison tool. This section details the implementation of dependency parsing based on the previous discussion. The main file in this stage of our system is `"FuncParser.y"`. Bison takes this file as input and produces a C file which will be the actual parser. Implementing `"FuncParser.y"` file focuses on three steps, which include writing the rules, semantic actions, and extra functions to extract dependencies and produce the output file. The following subsections explain these steps in more detail.

7.3.1 Definitions and Grammar Rules

As Flex and Bison are connected, all tokens are declared within the Bison declarations section. This section contains declarations that define terminal (also known as a token type) and nonterminal symbols. Fig 7.2 shows an example of this section taken from `"FuncParser.y"` file. For example, line 7 in this figure shows token type keyword in C (open bracket, close bracket, for and left bracket). The grammar rules section consists just of the grammar rules; but each individual rule consists of two parts: the BNF[2]syntax (e.g. `IDENTIFIER` in line 2 Fig 7.4) and the associated action to perform (e.g. the C statement in line 2 Fig 7.4), written as C code. Within the grammar rules section three collections of syntax are used. It contains grammar rules of C, C++ and the extra syntax of FLAME GPU all in BNF format. For BNF C grammar, the version that is based on the 2011 ISO C standard[3]was used. For preprocessing directives and C++ grammar, we used the C++ BNF Grammar that is available in [91] to cover all rules missing from the previous source. For the special FLAME GPU syntax, we added a part of the rules to process the grammar. The final BNF version can match both normal C and C++ code and any FLAME GPU functions. Fig 7.3 below shows part of the added rules that match agent function code. An example of an added rule, line 15, 16, 17 and 18 in Fig 7.3 consists of the macro definition of FLAME GPU function.

[2]Backus normal form (BNF) is a notation technique for context-free grammars (CFG), used to describe the syntax of programming languages.

[3]The rule section of Bison or(Yacc) specification of ANSI C grammar: http://www.quut.com/c/ANSI-C-grammar-y-2011.htmll

```
%{
C declarations...
%}
Bison declarations...

//Terminal symbols (tokens)
%token OPEN_BRACE  CLOSE_BRACE FOR LPAREN
//Terminal symbols (another type of tokens)
%type <ival>  IDENTIFIER I_CONSTANT STRING_LITERAL
//Nonterminal symbols (used in grammar rules)
%type <tval>  primary_expression postfix_expression

%%
 Grammar rules...
%%

```

Figure 7.2: A part of the Bison declaration section shows token types.

```
program:..
  .
  .
function_specifier
: INLINE
| NORETURN
| __FLAME_GPU_FUNC__            //Macro definition for all FLAME GPU functions
|__FLAME_GPU_INIT_FUNC__        //Macro definition for initialisation functions
|__FLAME_GPU_STEP_FUNC__        //Macro definition for step functions
|__FLAME_GPU_EXIT_FUNC__        //Macro definition for exit functions
;
xmachine_specifier
: XIDENTIFIER '*'                      //xmachine_memory_AgentName*
| XLIDENTIFIER '*'                     // xmachine_memory_AgentName_list*
| MLIDENTIFIER '*' primary_expression // xmachine_message_MessageName_list*
| MIDENTIFIER '*'                     //xmachine_message_MessageName*
|XMPIDENTIFIER '*'                     //xmachine_message_MessageName_PBM*
;
```

Figure 7.3: An example of some grammar rules that have been used to match FLAME GPU syntax.

7.3.2 Rule Actions

Each grammar rule can have an action written using C statements. This action is executed every time the parser recognizes a match for that rule. In our parser, we used more than one function call in some rules to complete the action. We create a new node for the parsing tree by calling `create_inode()` function. This function applies to each rule in the grammar to build the parsing tree for the source code and to hold the semantic value. Beside calling `create_inode()` function to create a new node, for specific rules, the `installId()` function needs to be called to extract some information about the variable of agent type dependency. This information will be processed and

used for generating the output file. Fig 7.4 shows a part of the grammar rules from Funcparser.y file. Rules appear on the left side, and each rule ends with one action using `create_inode()` function. For `MNIDENTIFIER` rule, for example, we use two functions within the action. As this rule representing a specific syntax, the `installId()` function (appears in line 7 in Fig 7.4) helps to hold values for later processes.

```
1  primary_expression
2  : IDENTIFIER          {$$= create_inode($1,PRIMARY_EXPRESSION, NULL, NULL, NULL, NULL);}
3  | constant            {$$= create_inode(NOTHING,PRIMARY_EXPRESSION, NULL, NULL, NULL, NULL);}
4  | string              {$$= create_inode(NOTHING,PRIMARY_EXPRESSION, $1, NULL, NULL, NULL);}
5  | '(' expression ')'{$$= create_inode(NOTHING,PRIMARY_EXPRESSION, $2, NULL, NULL, NULL);}
6  | generic_selection {$$= create_inode(NOTHING,PRIMARY_EXPRESSION, $1, NULL, NULL, NULL);}
7  |MNIDENTIFIER         {installId($1,MNIDENTIFIER , SAME, "MESSAGE","nothing");
8                       $$= create_inode($1, PRIMARY_EXPRESSION,NULL, NULL, NULL, NULL);}
9
10
```

Figure 7.4: A part of the grammar rules from our parser file with the rule's action.

7.3.3 User Subroutines

The last section of the bison input file consists of the main and the definitions of `create_inode()` and `installID()` are supplied here, along with other semantic functions. The rest are used to generate the final output code. Within the main function, the `yyparse()` function, which initiates the parser, will be called. This function is responsible for reading tokens, executing actions, and ultimately returns with a value of 0 when it encounters end-of-input or with a value of 1 when parsing failed because of invalid input (syntax error).

7.3.4 Generating the Meta Data Output File

During the parsing process, the `installId()` function added some remarks to specific symbols within the symbol table. These remarks help with other methods we have created to look for dependent data in each function. Certain agent variables may be read, or written by an agent function and we seek to identify these and make this explicit in a special output file in XML format. For generating the final output, we create two methods to produce the final results using both the AST and symbol table. The `WriteXML()` function writes the header of the output file and calls the `print_dependency` function to print out the data and message dependency of each agent function that has been

found in the `functions.c` file. An example of the input code is shown in Listing 7.2, and the output is shown in listing 7.3. Listing 7.2 shows a basic example of a FLAME GPU function. This function is taken from the Circles model 8.2; this model has one type of agent called a circle. This agent has six memory variables. However, the `move()` function only needs to access 2 memory variables for reading and 2 variables for writing. listing 7.3 views the dependency information in the `move()` function which has been written in an XML format to combine it easily with the original `XMLModelFile.xml` file at a later stage.

```
__FLAME_GPU_FUNC__ int move(xmachine_memory_Circle* xmemory)
{
 xmemory->x += xmemory->fx;
 xmemory->y += xmemory->fy;
 return 0;
}
```

Listing 7.2: The body of function move (circle model)

```
</functions>
      <function>
          <name>move</name>
          <xagent><name>Circle</name></xagent>
          <In_data>true</In_data>
              <In_datadependency>
                  <variable><name>fx</name></variable>
                  <variable><name>fy</name></variable>
               </In_datadependency>
           <Out_data>true</Out_data>
              <Out_datadependency>
                  <variable><name>x</name></variable>
                  <variable><name>y</name></variable>
              </Out_datadependency>
        </function>
</functions>
```

Listing 7.3: In and out meta-data dependency extracted from function Move (circles model)

7.4 Data Aware Simulation

In the previous sections, the implementation of the dependency generator was described, along with tools that have been used to build it. The dependency generator can parse any `functions.c` file (that holds the scripted agent functions), and produces an XML file consisting of messages and data dependencies between agent functions. These data will be used to establish the process of applying the data-aware approach.

Based on the formal definition of a communicating X-Machine (CXM) that was described in chapter three 3.1.1 earlier, the smallest unit that can be processed by current FLAME GPU is an agent. Whenever agents communicate with each other, all agent memory variables need to be copied for reading and writing to the memory. Moreover, this happens in every transition function from one state to another as seen in Fig 7.5. As such automating variable level data aware optimisation requires a new representation in which the smallest unit of dependency is at the variable not agent level.

This section proposes an alternative representation of CXM model in which individual units of m (members of the memory set M) for each agent function (φ in Φ, in the formal definition) that can communicate using a subset of data in the messages list r (members of the communicating relation set R). Focusing on a subset of this data will minimise the data movement and remove unnecessary memory movement. Fig 7.6 shows the main idea of our proposed method. Instead of reading and writing all agent memory in every state transition, the focus will be on the dependent data of each function (the subset of agent memory that has been used within each function). Extracting data and message dependencies of agents functions is the key to solving this issue. In the previous chapter, an automated process was described which uses the flex and bison tools to parse agent behaviour functions (C code) from FLAME GPU and produces a meta-data description of all dependencies between transaction functions. The following sections discusses the implementation of applying variable level data aware movement. Fig 7.7 summarises how the proposed method is linked to the current FLAME GPU SDK. In this figure, which describes the processing stages used to create the FLAME GPU runtime, the existing data path is shown by purple arrows and the data path used in our new pre-processing stage is shown

with red arrows. The next section discusses the implementation of merging dependency meta-data with a FLAME GPU model description and section 7.6 demonstrates the necessary modifications to the FLAME GPU templates to provide optimised data movement.

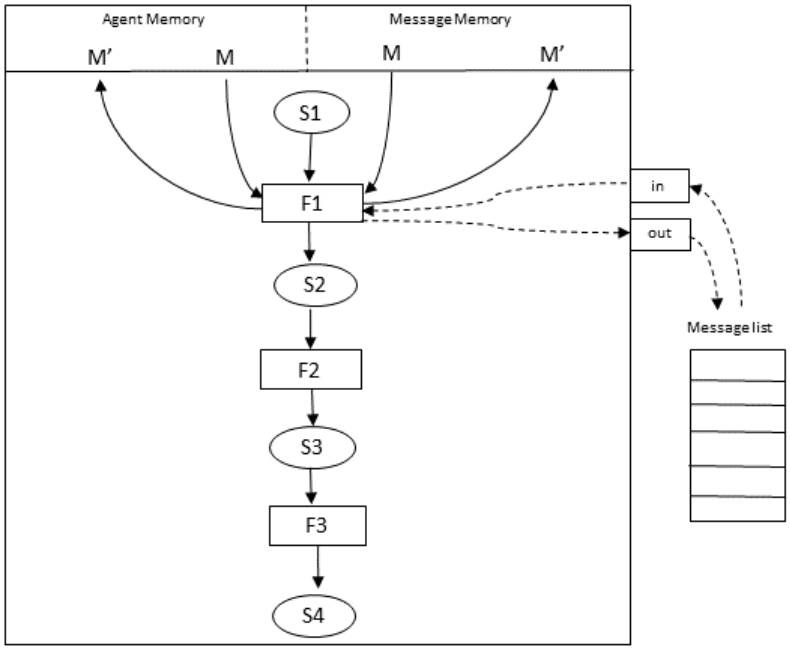

Figure 7.5: Stream X-Machine Specification, M and M' represent the agent memory set before and after agent function F1 which inputs and outputs messages to the message list [126]

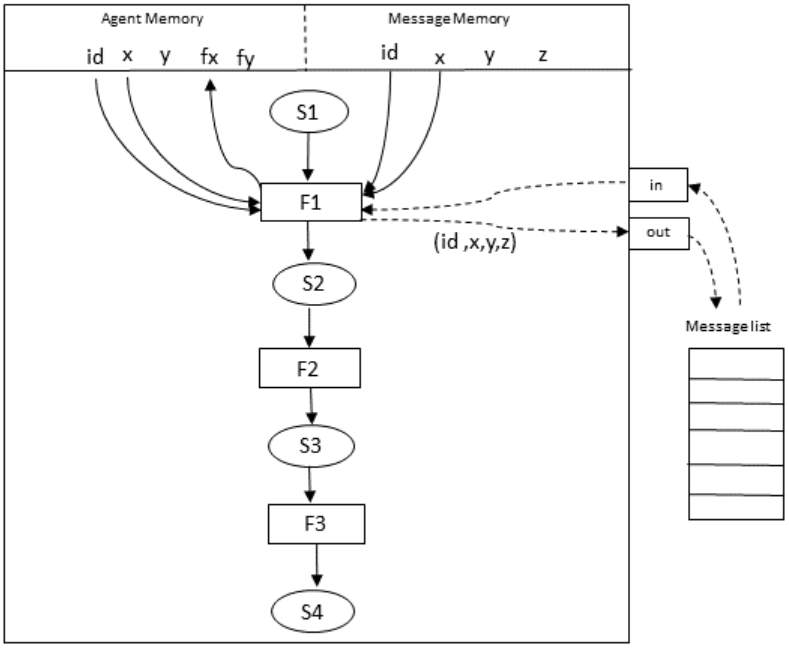

Figure 7.6: The smallest unit that can be processed through transition function is individual variables of agent memory instead of an agent's full memory set to minimise data movement.

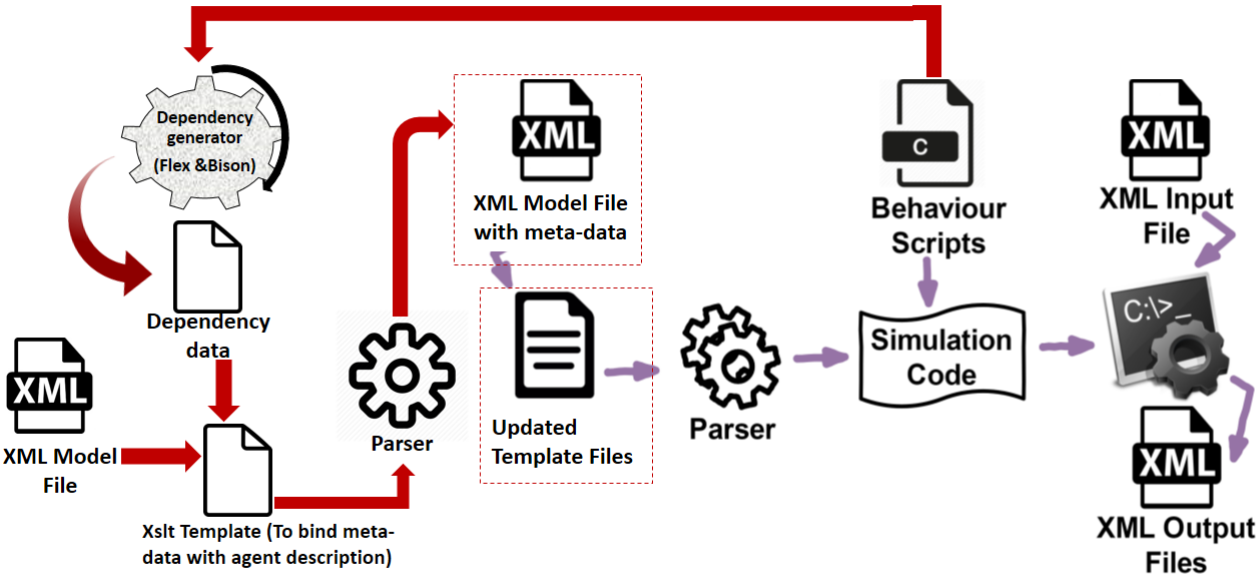

Figure 7.7: Processing stages used to create the FLAME GPU runtime, showing original (purple) and additional (red) data paths.

7.5 XSLT-Transformations for Merging Meta-data with Agent Descriptions

The first step in merging variable level meta data with modell descriptions starts from the dependency meta data generation in the previous chapter. The dependency parser generates an XML file which consists of function names and the `In_data`, `Out_data`, and `In_messages` within each function. The second step described in this section is to generate a new annoted `XMLModelFile.xml`. This can be achieved using an (XSLT) template. This processing stage combining the original `XMLModelFile.xml` with the meta-data requires file from the first stage. The output file contains the original model with meta information describing data dependencies. The following subsections explain this process providing some examples.

7.5.1 Input Files and Output Files of Merging Metadata Process.

Two XML files (`DataDependency.xml` and `XMLModelFile.xml`) and an XSLT template are the input files required to update the model specification file. The root element of the meta-data file (produced from 7.3.4) called `Data Dependency.xml` is `<function>`. This element consists of at least one element called `<function>` which includes an agent's name as attribute and all dependency types that might be found during the parsing stage. The second input file required to merge meta data with agent description is the original `XMLModelFile.xml` file. This file consists of the model specification that has been explained within chapter 3. The essential element of this XML model description file is `<function>` which will be our target to insert the meta-data as child elements once the function name matches. Table 7.3 summarises the XML elements that may be inserted into the annoted model output file. To perform the merge an XSLT template is described within the following subsection. An overview of the complete merge process can be seen in Fig 7.8.

7.5.2 The XSLT Template

As described earlier in chapter 2 section 2.4.1, XSLT is a flexible language that is used to translate XML documents to other formats such as HTML or other

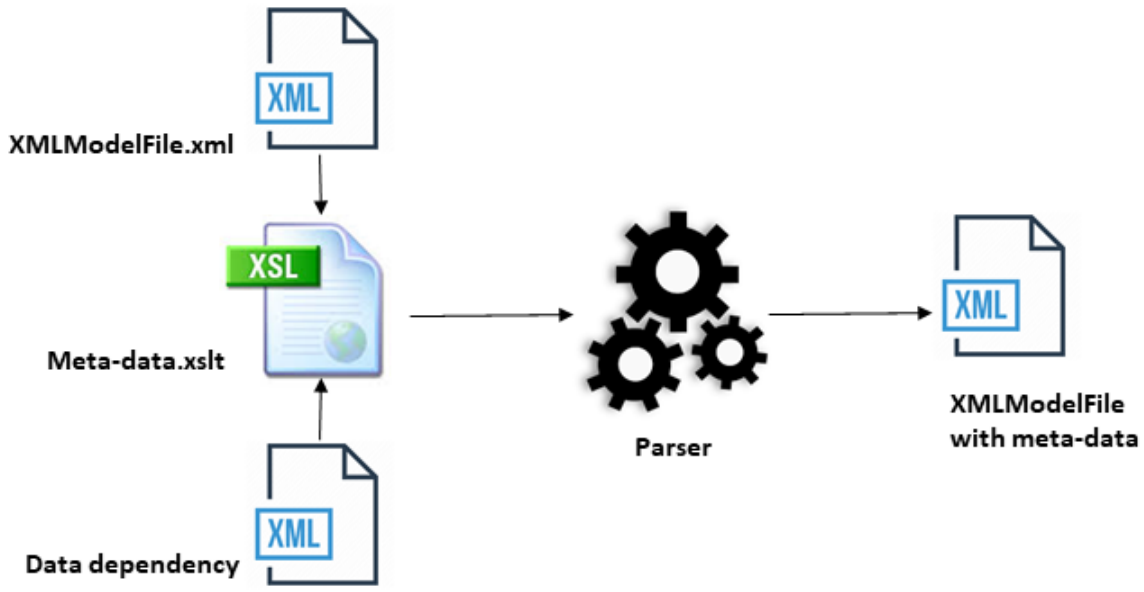

Figure 7.8: Input files and output result of merging data dependency with model specification using XSLT processor (msxsl.exe)

document formats. An XSLT template has been designed to generate a new version of the model description which contains the additional meta data information.

The XSLT processor recursively matches XML nodes of both input files and applies a template to it. In our case, the template needs to match nodes that hold the function's name. The name of the XSLT template that has been designed is `Meta-data.xslt`. Once the processor matches a node, `(Meta-data.xslt)` it will apply the following steps to generate the output file:

- Print out any elements which may occur before our new dependency elements within parent element `gpu:function`. This includes the name of the function, current states and the next state. The original description of the function including any of the original elements.
- Output any elements for the dependency elements and this include in_dependency, out_dependency and message_dependency elements using as shown in Fig 7.9.
- Output any elements which may occur after the new dependency elements. This includes all cases of input/outputs/conditions/ global conditions.

The code samples below (from `Meta-data.xslt`) demonstrates how

Table 7.3: The additional elements over the current FLAME GPU XML schema

XML Element	Parent Element	Description
<in_datadependency>	</gpu:function>	Holds any number of <dependencyVariable> elements of in_data dependencies of the function.
<out_datadependency>	</gpu:function>	Holds any number of <dependencyVariable> elements of out_data dependencies of the function.
<in_messagedependency>	</gpu:function>	Holds any number of <dependencyVariable> elements of in_message dependencies of the function.
<dependencyVariable>	<in_datadependency>or <out_datadependency>or <in_messagedependency>	Defines the <name> of dependency Variable within each type.

the iterative `for-each` element is used to generate dependencies for each `function_name` within `DataDependency.xml` file (if there are any). The template uses conditional statements to query if there is any data dependency within each function. The complete code of our template `(Meta-data.xslt)` can be found in Appendix D

7.5.3 Results

To demonstrate the flexibility of the approach the circle model is used. this model is widely used as a FLAME GPU benchmark and demonstrate that our technique is applicable to any FLAME GPU model. Fig 7.10 shows an example of the dependency data that have been extracted from the function called 'Move' within the `Circles` model [128]. This function consists of in and out data dependency as shown in Fig 7.10: the left side (A) of the figure shows the original code of the model description while the right side (B) of the figure shows the model description after adding meta-data. Both the dependency file

```
<!-- Output ANY elements for our new in_dependency elements -->
<xsl:for-each select="$lookup-source/functions/function[name=$function_name]">
<xsl:if test="In_data='true'">
    <in_datadependency>
    <xsl:for-each select="In_datadependency/variable">
        <dependencyVariable><name>
        <xsl:value-of select="name"/>
        </name></dependencyVariable>
    </xsl:for-each>
    </in_datadependency>
    </xsl:if>
</xsl:for-each>
```

Figure 7.9: The XSLT template that generates in_data dependency for each function. Same loop can be applied to Out-data and In-message to produce respective Out_dependency, In_message dependency elements.

and the model description file with mate-data can be found in Appendix D.

Figure 7.10: A: A part of the Circles model description showing function 'move'. B:The model description after adding meta-data. C: The actual body of the function 'move' from functions.c file

7.6 FLAME GPU Template Files

Chapter 3 discussed the mechanism of data movements and internal communication during simulation of FLAME GPU model and how FLAME GPU deals with the process of reading and writing data in global GPU memory. This section demonstrates how the FLAME GPU templates 3.3.4 can be modified to utilise annotated model descriptions to access required data only. The effect of which minimised data movement during simulation.

FLAME GPU generates simulations by applying these templates to the model files which is linked with the behaviour scripts to generate a simulation program. All agent and message memory will be accessed during this process using fast caches, shared memory for agent variables and texture memory for message variables. With the proposed method the templates have been modified to access only required data for both agent and messages. Two of the FLAME GPU templates have been modified `FLAMEGPU_kernels.xslt` and `simulation.xslt`. `FLAMEGPU_kernels.xslt` template. The output of `simulation.xslt` is a source file containing the host side simulation code which includes loading data to and from the GPU device and making a number of CUDA kernel calls which perform the simulation process. To access a subset of agent memory for reading and writing a part of the code in `FLAMEGPU_kernels.xslt` has been modified to customise memory access. The original template (that is accessing all agent memory) is shown in Fig 7.11 while Fig 7.12 shows an example of the updated template code accessing a subset data for both reading and writing memory. In Fig 7.11, this template is targeting all agent memory variables (appears in red boxes) to generate a code (C structure which contains a member variable for each agent memory variable) that allows memory access pattern for both reading and writing data in global GPU memory. In Fig 7.12 the template specifically used the dependency data in each function (appears in red boxes) to generate the data movements (C structure which contains a member variable for each function dependency variable) .

For accessing a subset of message variables both templates above are modified. For iterating message lists (message input)

within agent functions, there are two functions are provided for each message list `get_first_*name*_message(args...)` and `get_next_*name*_message(args...)`. The templates to allow these function to access the required memory for each message were modified. In the current version of the FLAME GPU templates the message variables are copied to the texture memory by default whereas the updated version copies only the dependent message variables, and initialises all unused values to 0. An example of both versions of XSLT output can be seen in listing 7.4 for the current version and listing 7.5 for the updated version. An example of the code that accesses the message dependencies taken from `simulation.xslt` template appears in Fig 7.13. Both templates which have been updated are included in full within Appendix D.

```
1
2   //Using texture cache in the old version, copy all message
      variables by default
3 temp_message.id = tex1Dfetch(tex_xmachine_message_location_id,
      cell_index + d_tex_xmachine_message_location_id_offset);
4 temp_message.x = tex1Dfetch(tex_xmachine_message_location_x,
      cell_index + d_tex_xmachine_message_location_x_offset);
5 temp_message.y = tex1Dfetch(tex_xmachine_message_location_y,
      cell_index + d_tex_xmachine_message_location_y_offset);
6 temp_message.z = tex1Dfetch(tex_xmachine_message_location_z,
      cell_index + d_tex_xmachine_message_location_z_offset);
```

Listing 7.4: The generated code from the current template which is coping message memory by default.

```
//SoA to AoS - xmachine_memory_<xsl:value-of select="xmml:name"/>
 Coalesced memory read (arrays point to first item for agent index)
 xmachine_memory_<xsl:value-of select="../../xmml:name"/> agent;
 <xsl:for-each select="../../xmml:memory/gpu:variable">
 <xsl:choose><xsl:when test="xmml:arrayLength">agent.
 <xsl:value-of select="xmml:name"/> = &amp;(agents-&gt;
 <xsl:value-of select="xmml:name"/>[index]);
 </xsl:when>
 <xsl:otherwise>
 agent.<xsl:value-of select="xmml:name"/> = agents-&gt;
 <xsl:value-of select="xmml:name"/>[index];
 </xsl:otherwise>
 </xsl:choose>
 </xsl:for-each>


//AoS to SoA - xmachine_memory_<xsl:value-of select="xmml:name"/>
 Coalesced memory write (ignore arrays)
 <xsl:for-each select="../../xmml:memory/gpu:variable">
 <xsl:if test="not(xmml:arrayLength)">agents-&gt;
 <xsl:value-of select="xmml:name"/>
 [index] = agent.<xsl:value-of select="xmml:name"/>;
 </xsl:if>
 </xsl:for-each>
```

Figure 7.11: The original XSLT template generating code accessing all memory.

```
//Data aware copy from agent memory
//SoA to AoS - xmachine_memory_<xsl:value-of select="xmml:name"/>
 Coalesced memory read (arrays point to first item for agent index)
 xmachine_memory_<xsl:value-of select="../../xmml:name"/> agent;
 <xsl:for-each select="xmml:in_datadependency/xmml:dependencyVariable">
 <xsl:choose><xsl:when test="xmml:arrayLength">
 agent.<xsl:value-of select="xmml:name"/> = &amp;
 (agents-&gt;<xsl:value-of select="xmml:name"/>[index]);
 </xsl:when>
 <xsl:otherwise>
 agent.<xsl:value-of select="xmml:name"/> = agents-&gt;
 <xsl:value-of select="xmml:name"/>[index];
 </xsl:otherwise>
 </xsl:choose>
 </xsl:for-each>


 //Data aware copy from agent memory
 //AoS to SoA - xmachine_memory_<xsl:value-of select="xmml:name"/>
 Coalesced memory write (ignore arrays)
 <xsl:for-each select="xmml:out_datadependency/xmml:dependencyVariable">
 <xsl:if test="not(xmml:arrayLength)">agents-&gt;
 <xsl:value-of select="xmml:name"/>
 [index] = agent.<xsl:value-of select="xmml:name"/>;
 </xsl:if>
 </xsl:for-each>
```

Figure 7.12: The modified XSLT template that generates code accessing required data only.

```
<xsl:variable name="function" select="."/>
<xsl:variable name="input_message_name" select="xmml:inputs/gpu:input/xmml:messageName"/>
<xsl:if test="xmml:inputs">
<xsl:for-each select="../../../../xmml:messages/gpu:message[xmml:name=$input_message_name]">

bool h_<xsl:value-of select="xmml:name"/>_message_read_deps[<xsl:value-of select="count(xmml:variables/gpu:variable)"/>];
memset(&amp;h_<xsl:value-of select="xmml:name"/>
_message_read_deps, false, sizeof(bool)* <xsl:value-of select="count(xmml:variables/gpu:variable)"/>);
<xsl:for-each select="xmml:variables/gpu:variable">
<xsl:variable name="variable_name" select="xmml:name" />
<xsl:if test="$function/xmml:inputs/gpu:input/xmml:in_messagedependency/xmml:dependencyVariable[xmml:name=$variable_name]">
h_<xsl:value-of select="../../xmml:name"/>_message_read_deps[<xsl:value-of select="position() - 1" />] = true;
</xsl:if>
</xsl:for-each>
gpuErrchk( cudaMemcpyToSymbol(d_<xsl:value-of select="xmml:name"/>_message_read_deps, h_<xsl:value-of select="xmml:name"/>
_message_read_deps, sizeof(bool)*<xsl:value-of select="count(xmml:variables/gpu:variable)"/>));
</xsl:for-each>
</xsl:if>
```

Figure 7.13: A part from the modified XSLT template that generates code accessing required message only.

7.7 Summary

This chapter explained the implementation of the dependency parsing system, which consists of three main stages: scanning the source code to generate tokens (lexical analysis), parsing the code to produce both the AST and the symbol table (syntax analysis and semantic analysis), and finally, generating the meta-data output file which consist variable dependency for each function. This chapter also focused on how to use dependency information to optimise the FLAME GPU models. By using the data dependency for each function, this chapter described the process of modifying FLAME GPU templates to access the required memory. In the next chapter, the evaluation of FLAME GPU performance is conducted through the use of a number of models to test the scalability, complexity and flexibility. It also presents the results by comparing the simulation execution time of both systems (the current FLAME GPU and extended FLAME GPU).

```
1
2 //Using texture cache in the updated version, copy only required
     variables
3 if(d_location_message_read_deps[0]){
4   temp_message.id = tex1Dfetch(tex_xmachine_message_location_id,
     cell_index + d_tex_xmachine_message_location_id_offset);
5 }else {
6 temp_message.id = 0;
7 }
8
9 if(d_location_message_read_deps[1]){
```

```
10   temp_message.x = tex1Dfetch(tex_xmachine_message_location_x,
       cell_index + d_tex_xmachine_message_location_x_offset);
11 }else {
12 temp_message.x = 0;
13 }
14
15 if(d_location_message_read_deps[2]){
16   temp_message.y = tex1Dfetch(tex_xmachine_message_location_y,
       cell_index + d_tex_xmachine_message_location_y_offset);
17 }else {
18 temp_message.y = 0;
19 }
20
21 if(d_location_message_read_deps[3]){
22   temp_message.z = tex1Dfetch(tex_xmachine_message_location_z,
       cell_index + d_tex_xmachine_message_location_z_offset);
23 }else {
24 temp_message.z = 0;
25 }
```

Listing 7.5: The generated code from the updated template which is copying the required variables from message memory.

Chapter 8

Results

8.1 Introduction

The previous chapter described the modification of the FLAME GPU framework to make it data aware and reduce memory movement. To evaluate the benefits of using the data-aware approach, this chapter shows a comparison of results between current FLAME GPU and the data-aware version using three different models. It is organised as follows, section 8.2 shows the comparison results between both systems using a simple force resolution model (referred to as the Circles model). Section 8.3 covers the benchmarking results of both versions using the benchmark model that was explained in chapter 5. Within section 8.4, the keratinocyte (cell) model is further used to evaluate the performance and demonstrates the advantages of the data-aware approach to a complex (non-benchmark) model used with systems biology research.

8.2 The Circle Model

The Circles model is a simple force resolution model that has been used to evaluate the simulation performance of different message communication techniques between FLAME and FLAME GPU [128]. This model consists of a single agent and message type with three agent functions. The first function to output agent location information through a message and the second function for reading location from the same message and the last function used to move the agent according to interagent repulsive forces. The agent memory consists of six variables, so using the current FLAME GPU, each function requires access to all variables to update agent memory. Using the proposed method will

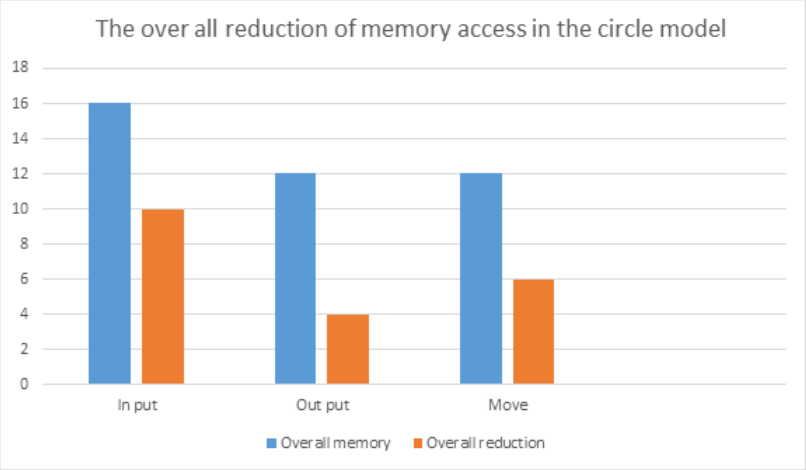

Figure 8.1: The total data movement reduction of each function within Circles model.

reduce memory access in each of function. Figure 8.1 and table 8.3 shows the total reduction of memory access within each function after implementing the model using data-aware FLAME GPU. The machine used for benchmarking both versions in this experiment uses NVIDIA TITAN V graphics card with 5120 CUDA cores and 12 GB HBM2 of memory. Figure 8.2 shows the performance of the Circles model using spatial partitioning of message communication in both versions. The proposed method (orange line) shows significant speed improvements when compared to current FLAME GPU (blue line). The performance measurements, in ms, are made by averaging the performance over 10 iterations at various population sizes.

Table 8.1: The total memory access for each agent function in the circle model and the percentage of reduction after applying our approach

Function:	Input	Output	Move
Memory access	12	12	12
In-data	5	3	4
Out-data	2	0	2
Total reduction	41%	75%	50%

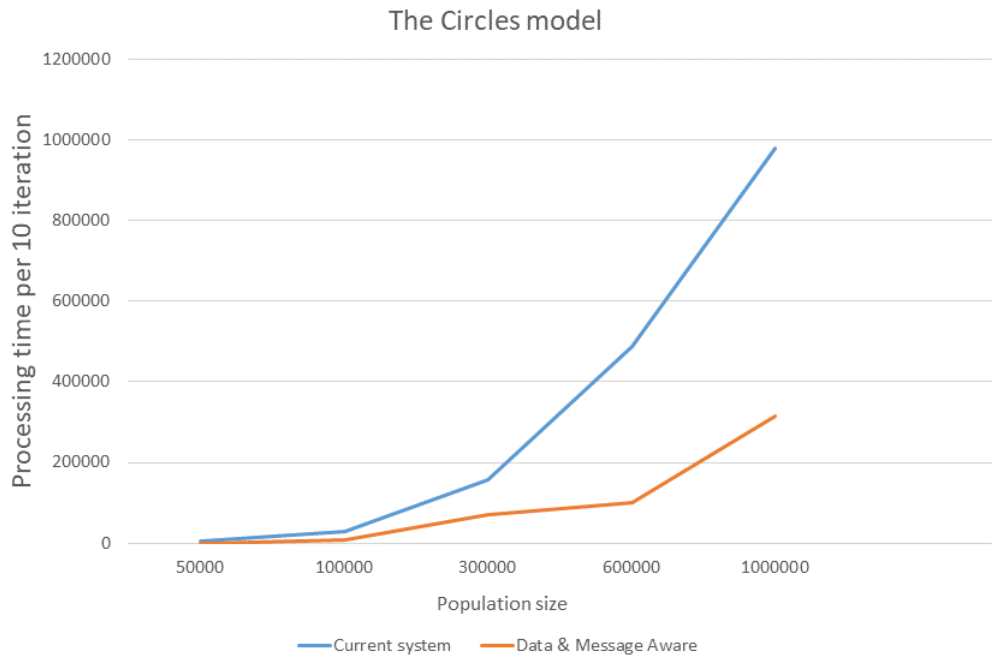

Figure 8.2: Comparison of average execution time against population size, show-ing unmodified (blue) and modified (orange) FLAME GPU.

8.3 Our benchmark model

The benchmark model (described in chapter 5) is based on the concept of particle-based chemical interaction simulation and accepts input parameters that control both system scalability, population and agent homogeneity. For system scalability, the population size for each agent type will be increased. In agent homogeneity, the focus will be on increasing the complexity for both individuals (by increasing communication) and the overall population (by increasing diversity of agent type). Three different benchmarks were used to examine the performance efficiency for both systems: scalability, divergence within the population and divergence within an agent. The machine used for benchmarking both versions has an NVIDIA TITAN Xp graphics card with 5120 CUDA cores and 12 GB of memory. Each experiment was run 30 times for each sample, and the median value of the execution time was calculated and presented in a graph for each experiment.

8.3.1 Scalability

This benchmark will measure the scalability of the performance of both systems. The population size of each agent type starts with 100,000 agents and

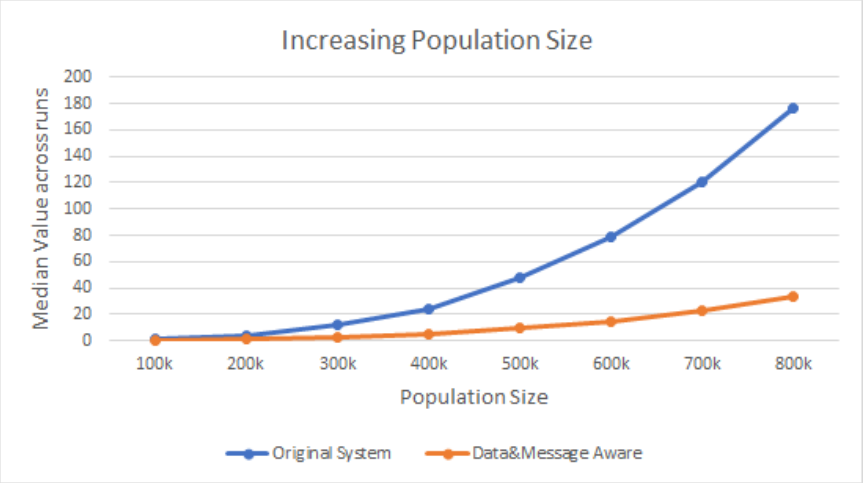

Figure 8.3: Comparison of average execution time against population size, showing unmodified (blue) and modified (orange) FLAME GPU.

ends with 800,000 agents. This benchmark is based on the same example that was described within section 5.4.3 in chapter 5. It is representative of scaling the model and the simulation was performed for 100 iterations. In figure 8.3 the proposed method (orange line) shows significant speed improvements when compared to current FLAME GPU (blue line). With population size equal to 300,000 and above the average of improvement reaches 80%. The interquartile range values for the data aware approach system of this experiment can be seen in figure 8.4.

Population	100k	200k	300k	400k	500k	600k	700k	800k
Medain	0.451352	1.218011	2.151542	5.119767	9.845623	14.77364	23.22051	33.88867
Q1	0.434529	1.212355	2.14171	5.095672	9.797689	14.76318	23.2035	33.82633
Q3	0.448088	1.229901	2.159507	5.116974	9.855426	14.80212	23.27358	33.90736
Min	0.428604	1.205446	2.132603	5.080378	9.769404	14.72824	23.16641	33.77504
Max	0.459536	1.243849	2.165822	5.13149	9.883079	14.84085	23.33769	33.97893
Range	0.030933	0.038403	0.033219	0.051112	0.113675	0.112605	0.171279	0.203887

Figure 8.4: Interquartile range values in seconds for simulation runs (while applying the functional approach and increasing the population size)

8.3.2 Divergence within the Agent

The main concept of this experiment is to observe the effects of divergent behaviour (within an agent) on the execution time for both systems. Based on the same example that is used by the benchmark model within section 5.4.2 in chapter 5, this benchmark is representative of increasing the individual complexity of an agent, and that means more functions in each layer every cycle. As

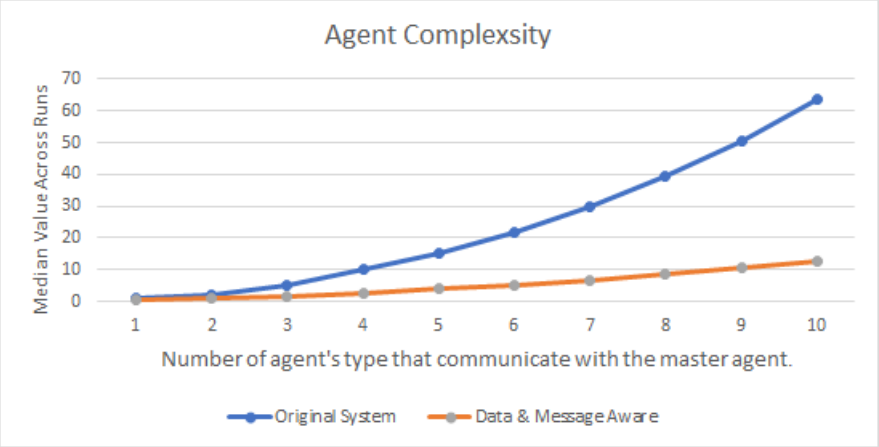

Figure 8.5: Comparison of processing time against number of communicating agents (slave-to-master), showing unmodified agents (blue) and modified (orange) FLAME GPU.

the function layers represent the control flow of simulation processes in FLAME GPU, adding more agent functions every time will increase the number of the layer in each cycle (as functions of the same agent need to be processed in sequential order) and that will lead to increasing the execution time of each iteration. The increase in execution time can be observed in figure 8.5 for both versions (current version with the blue line and modified one with orange line) with significant reduction of execution time when using the proposed system (as more agent functions provides more opportunity for optimisation. The average of improvement reaches 69%, and with more divergence within an agent, the data aware system showed more time reduction in simulation execution time compared with the current system. The population size that has been used in this benchmark is 100,000 for each type of agent, and each simulation was run for 100 iterations using the same environment size. Figure 8.6 shows the interquartile range cross-run values for the data aware approach system.

No Slaves	1	2	3	4	5	6	7	8	9	10
Median	0.451352	0.984864	1.738304	2.725387	3.94419	5.279242	6.841458	8.623511	10.55199	12.72256
Q1	0.434529	0.969195	1.73557	2.7085	3.942397	5.261428	6.838364	8.625128	10.55202	12.72364
Q3	0.448088	0.985605	1.752973	2.727062	3.950507	5.278172	6.855375	8.636381	10.56196	12.73497
Min	0.428604	0.951759	1.728562	2.691396	3.921439	5.240544	6.83256	8.600796	10.5458	12.70651
Max	0.459536	0.993425	1.79356	2.737208	3.960637	5.284387	6.861507	8.651791	10.56867	12.74768
Range	0.030933	0.041666	0.064999	0.045812	0.039198	0.043843	0.028947	0.050995	0.022866	0.04117

Figure 8.6: Interquartile range values in seconds for simulation runs (agent complexity benchmark while applying the functional approach)

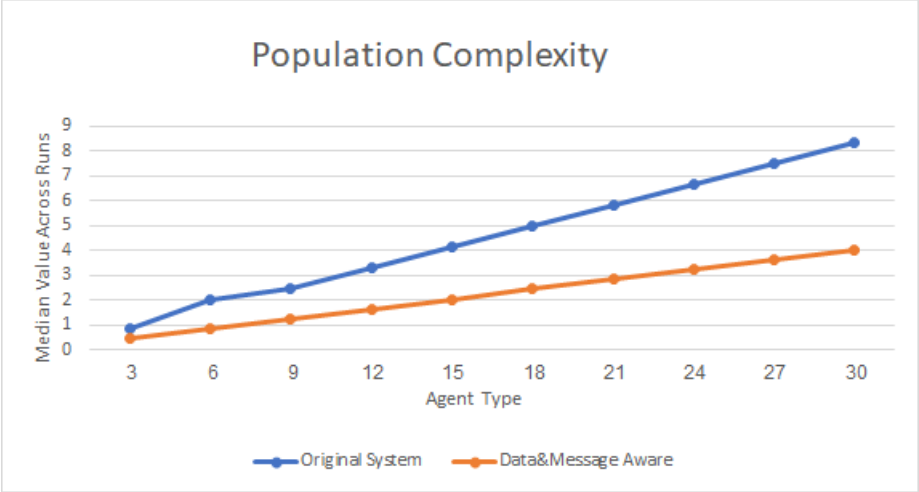

Figure 8.7: Comparison of median value of the execution time against population divergence, showing unmodified of using current (blue)and modified (orange) FLAME GPU.

8.3.3 Divergence within the Population

Observing the system performance while increasing population complexity will be the focus of this benchmark. This experiment starts with a simple model containing three types of agent, ten agent functions and three kind of message and ends with 30 agent types, 100 agent functions, and 30 message types. The execution time that has been performed in both systems can be observed in figure 8.7, the blue line represents the current FLAME GPU, and the orange line shows the system using the data aware approach. The graph has linear performance as increasing agent types has a linear increase agent functions within the model. The interquartile range values cross-runs for the data aware approach system can be seen in Fig 8.8. Each simulation in this benchmark was run for 100 iterations using the same environment size and the population size of each agent type was 100,000 agent.

Agents	3	6	9	12	15	18	21	24	27	30
Median	0.451352	0.845799	1.227567	1.626204	1.997808	2.443637	2.835021	3.230114	3.635973	4.016735
Q1	0.434529	0.838358	1.229249	1.610337	1.993008	2.425371	2.823644	3.219983	3.633934	3.998098
Q3	0.448088	0.84983	1.240903	1.621229	2.009706	2.444365	2.83799	3.236833	3.643817	4.027211
Min	0.428604	0.832739	1.219764	1.590813	1.985754	2.415955	2.813828	3.207859	3.618375	3.986311
Max	0.459536	0.870976	1.249061	1.639244	2.019584	2.456508	2.846907	3.251484	3.685608	4.036517
Range	0.030933	0.038237	0.029297	0.048432	0.03383	0.040552	0.033079	0.043625	0.067233	0.050207

Figure 8.8: Interquartile range values in seconds for simulation runs (Population complexity benchmark while applying the functional approach)

8.4 The keratinocyte (cell) model

As part of the Epithelium project [156], the Keratinocyte (cell) model [142] was developed to model the behaviour of various types of skin epithelial cells. This model is used to observe the interactions between cells during normal and abnormal tissue growth. The simulation of Keratinocyte model represents some different stages within the cell cycle. This includes the renewal of cells by growing and dividing, which ultimately leads to increased agent population within the model. There is also differentiation, the behaviour here represents the change from Keratinocyte stem cells to fully specialised cells that can no longer replicate. Once the specialised cells reach their limit, they begin to die and the process of apoptosis eliminates such agents from the simulation.

Based on the model functionality that is described by Sun et al. [142], the Keratinocyte model has been implemented in FLAME GPU with some modifications [129]. It is also included within FLAME GPU SDK as one of the example models and provides an opportunity to explore the result of our data aware approach on a real "research" model. The modified version of the model within FLAME GPU consists of a single agent type, two message types, a single initialisation function and seven agent functions. There are four cell types implemented as different type values assigned to individual Keratinocyte agents which are as follows.

- Stem Cells: are existed at cell colonies and divide to produce two stem cells. They remain fairly static during the simulation.
- Transit Amplifying: same as stem cells, will divide to produce two transit amplifying cells, if there is enough space.
- Committed Cells: both stem and transit amplifying cells become committed cells after the differentiation processes.
- Corneocyte Cells: When a cell in any of the three above types dies it becomes a corneocyte cell.

Agent functions are used to simulate the biological processes of cell behaviour such as cell-cell and cell-substrate adhesion, migration, division and differentiation. Table 8.2 summaries all agent functions and their role within the model.

Table 8.2: Agent functions used within the Keratinocyte colony model.

Function Name	Description
output_location	Outputs the cell location
cycle	Simulates a cell cycle to allow the division process after a predetermined time.
differentiate	Simulates the differentiation process
migrate	Simulates cell movement(cell migration).
death_signal	Decides if a cell should become a Corneocyte cell.
force_reso lution out-put	Outputs the cell location after the normal simulation process of the cell.
resolve_forces	Cell spatial location overlaps are resolved by this function.

8.4.1 Performance Results

This experiment uses the original version of Keratinocyte model that is included with FLAME GPU examples. There is an updated version by Chimeh and Richmond [22] which is designated to reduce the impact of divergence within the original model. The original model has been chosen to observe the impact of data ware approach on overall performance for such complex behaviour. The amount of data and message reduction as a result of using the data-aware approach can be seen in table 8.3 agent memory and table 8.4 for message memory with each function. All results were obtained on a single PC using NVIDIA TITAN V graphics card with 5120 CUDA cores and 12 GB HBM2 of memory. Figure 8.9 shows the performance of the Keratinocyte model in both versions. The proposed method (orange line) shows miner improvements when compared to current FLAME GPU (blue line). This due to the complex behaviour of agent communication within `resolve_forces` function as the execution time of this function takes %90 of the iteration run time. There is a relationship between the average of message memory reduction and the average of the overall improvements when using this approach. The simulation was performed for 500 iterations each time while scaling the model. Averages for each experiment were obtained over 10 runs per sample.

Table 8.3: The total memory access for each agent function in the Keratinocyte model and the percentage of reduction after applying our approach

Function Name	Memory access	In-data	Out-data	Total reduction
Output_location	36	7	0	80%
Cycle	36	8	2	70%
Differentiate	36	8	2	70%
Death_signal	36	4	1	86%
Migrate	36	5	4	75%
Force_resolution_output	36	5	0	86%
Resolve_forces	36	7	4	69%

Table 8.4: The total memory access for each message in the Keratinocyte model and the percentage of reduction after applying our approach

Function Name	Message Name	Memory Access	In-data	Total reduction
Differentiate	location	9	4	55%
Death_signal	location	9	4	55%
Migrate	location	9	4	55%
Resolve_forces	force	5	4	20%

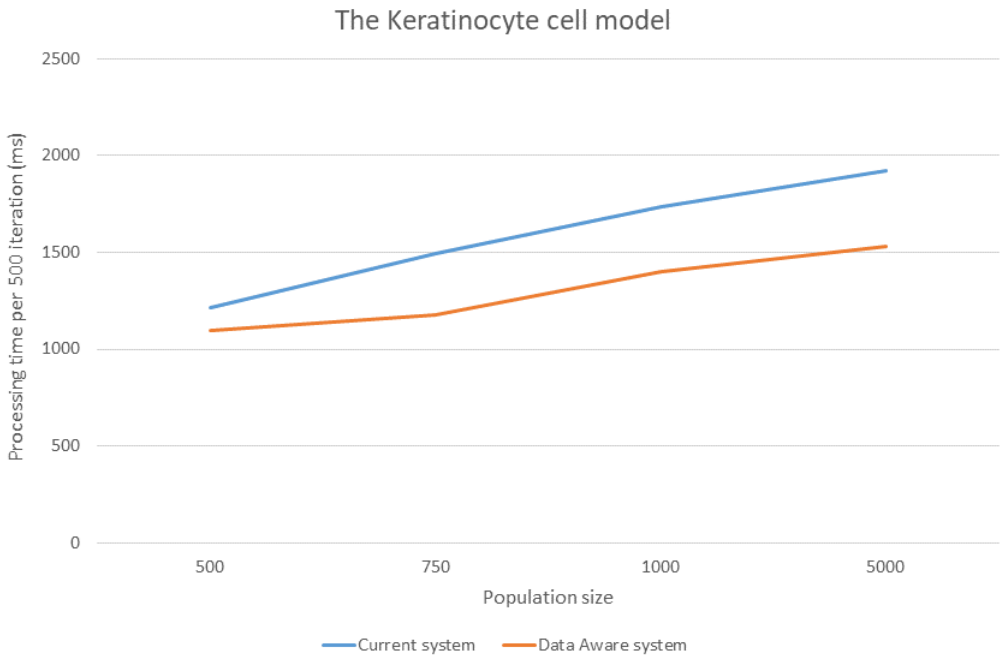

Figure 8.9: Comparison of average execution time against population size, showing unmodified (blue) and modified (orange) FLAME GPU.

8.5 Validating the Results

Within the FLAME GPU, the generated simulation header file (header.h) contains multiple macros that can be changed to output to console additional population data, for example, agent count per iteration or effects of more particular timing. Validating the results of both the Circle model and Keratinocyte model has been done using the output files, consisting of all the information about each iteration. For validating the benchmark model results, as mentioned in Chapter 4, the generated histogram of agent accounts was used to compare all of the experiments to ensure that the agent's behaviour was the same for both systems.

8.6 Discussion

The overall trend in performance of applying the data-aware technique to FLAME GPU is an improvement in operational efficiency. The rate of improvements is between 47% to 80% based on the type of variable (either agent or message) which has benefited from reduced data movement. The results indicate that FLAME GPU is most likely to benefit from reducing movement of message variables. This is can be observed from all benchmarking results in section 8.2, 8.3.1 and 8.3.2. This results observed within the result of modelling Keratinocyte model using are lower than that of the benchmark model (i.e. 20% improvement). This can be attributed to the behaviour of the mode. Within the Keratinocyte model a recursive force resolution step accounts for a large percentage of the run-time. During this step the `force` message reading is the dominant part of the execution time and as such the 20% overall performance improvement can be expected from the outputs of the dependency analysis. More generally the dependency analysis outputs which indicate the total reduction in data movement can provide a good estimate of simulation performance improvement when they are considered alongside profiling results of the model to understanding which parts of the model are dominant with respect to run-time. In addition to the results demonstrated an additional benefit to the proposed approach is that the reduction in data movement reduces register and shared memory usage which are used within FLAME GPU to cache agent and message variables. As registers and shared memory are an

limited resource the data-aware approach facilitates a modeller to design larger models (i.e. more variables) without exceeding resource limitations. Overall our results of using data-aware approach offer efficient performance for ABM application which is similar to performacne results reported for other types of run-time analysis 2.5.1.3 such as StarPU, PaRSEC and DAGuE. The distinction between other approaches is that they use a task based approach to reschedule the tasks to available resources rather than using prior knowledge of memory variable dependencies to reduce the amount of memory usage.

8.7 Summary

This chapter has presented the performance results of using both (original and data-aware) versions of the FLAME GPU to implement the simulation of three different models. The Circles model and the Keratinocyte model have been used to evaluate the benefits of applying a data-aware approach to FLAME GPU. The benchmark results in both models proved that accessing a subset of data memory reduces the execution time of the iteration. However, the percentage of performance enhancement is based on the behaviour complexity of each model. In our benchmark model, three benchmark experiments have been used to evaluate the overall performance of the new system. These experiments focused on measuring the ability for the new system to reduce simulation execution time under specific criteria (scalability and system homogeneity). Comparing the benchmark results of the current and new system show that reducing data movement within CS simulation improves overall performance. The scaling population size experiment for both systems showed that the new method helped to reduce execution time by approximately 80% and tends to stabilise around this percentage as population size increases. A significant improvement has resulted from using the proposed method within the divergence benchmark. Execution time was reduced by 70% when running the agent divergence benchmark and by around 45% while examining the population divergence experiment.

Chapter 9

Conclusion

This thesis has investigated the impacts of discovering data dependency to reduce data movement during the simulation of complex systems on a GPU. A new benchmark model has also been designed to examine system scalability, system homogeneity and the ability of the system to handle an increase of agent communications within ABM systems.

9.1 Research Summary

The work presented in this thesis investigated the nature of the ABM system, starting with developing a standard method of benchmarking ABM applications under a number of criteria for observing system performance. Measuring system scalability is the most common benchmark that exists in the field of ABM and simulation. In fact, the ability of such systems to scale is not the only factor that may affect the performance in these systems, according to the OpenAB community. The computational complexity, internal memory requirements and homogeneity of the agent and population of the model are other factors that may affect the overall performance. A standard benchmark model has been designed to allow for observing system behaviour while testing these factors. The performance results obtained by using the presented benchmark model, as discussed in Chapter 5.4, indicate that the presented benchmark model is suitable for use as an experimental tool in the evaluation of modelling capabilities of an ABM system if it is replicated in a suitable way. It also indicates that varying the population size is not the only factor that may slow the execution time of a simulation. This thesis also proved that all of the above factors have a clear impact on the performance of ABMS applications.

For the effect of data dependency on the performance of real-time applications, this thesis contends that the discovery of data dependency between agents could help to reduce data movement during simulations. This ultimately improves performance and reduces the execution time. By applying two proposed approaches for reducing data movement using extracted dependency, this thesis proves that the execution time is reduced as a result of data movement reduction. The approaches that were used are as follows:

9.1.1 Functional Approach

In a FLAME GPU, agent functions are organised into layers according to when their state conditions are enabled. Prior knowledge of the functional dependency helped to minimise the number of layers, leading to the minimisation of execution time per iteration of the simulation. This approach, based on merging some agent functions that have no dependency conflicts, was experimentally tested using the benchmark model with the FLAME GPU.

9.1.2 Data-Aware Approach

With FLAME GPU, all agent and message memory needs to be accessed during the simulation process. The use of limited shared memory and registers results in limitations on both the size of messages and the complexity of agent behaviour. This thesis presented a data-aware approach based on the use of data dependency information to access a subset of agent and message memory during the simulation. The data-aware approach was tested using a number of existing FLAME GPU models in addition to our benchmark model. The automation process of this approach is a result of implementing the following points:

- **Automating the discovery of data dependency** A dependency generator tool was created to easily parse the behavioural function scripts of an agent and produce an XML file detailing the discovered data dependencies. The XSLT processor was used to generate a new XML model file with extra meta-information by combining the model specification file with dependency data.

- **A new representation of an agent in the X-machine model** An alternative representation of a CXM model was proposed, and the proposed modification to the X-machine model allowed access to a subset of agent memory within each function. The modification to FLAME GPU templates to access the required memory was described.

9.1.3 Evaluating and Validating the Use of the Proposed Approaches

Applying both approaches, the performance of the system was evaluated through careful benchmarking against the original FLAME GPU. Figure 9.1 shows the comparison of the median value cross-runs of execution time for all versions. This benchmark measures the scalability of the model of each version and observes the performance, which is the same as reviewed in Section 5.4.3. The agent complexity experiment discussed in Section 5.4.2 obtained results that are presented in figure 9.2, which shows the median value of the processing time in relation to the number of communicating agents (slave-to-master). The performances of all systems were compared using the population complexity benchmark referenced in Section 6.5.1, and the result is presented in figure 9.3.

For all experiments, the original FLAME GPU (blue), data-aware approach (orange) and functional approach (grey) resulted in an average performance enhancement of 10–20% when using the functional approach, while the data-aware approach reached 80–90%. For validation, as mentioned in Chapter 4, the generated histogram of agent accounts was used for comparison in all of the experiments to ensure that the agent's behaviour was the same for every system.

In conclusion, both approaches showed a substantial improvement as compared with the original FLAME GPU, but the extent of this enhancement depended on the complexity of the model and the amount that data movement can be reduced. Table 9.1 shows the comparison of the approaches.

Table 9.1: Comparison of Data-aware approach and Functional approach

Criteria	Data-Aware approach	Functional approach
Implementation	Automatically	Manually
Targeting all agent functions to reduce agent memory access	All	Only functions have been merged
Can reduce the message memory access	Yes	Has no effect
Tested using a number of FLAME GPU existing models	Yes	Only benchmark model
Can be applied to other ABM simulation tools	Yes	Specified for FLAME GPU

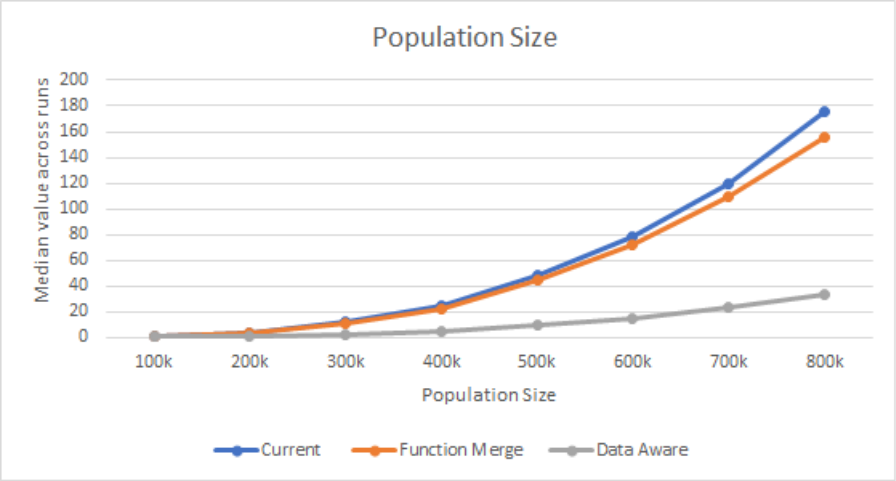

Figure 9.1: Comparison of the median value of execution time against the population size, showing the original FLAME GPU (blue), using the data-aware approach (orange) and the functional approach (grey).

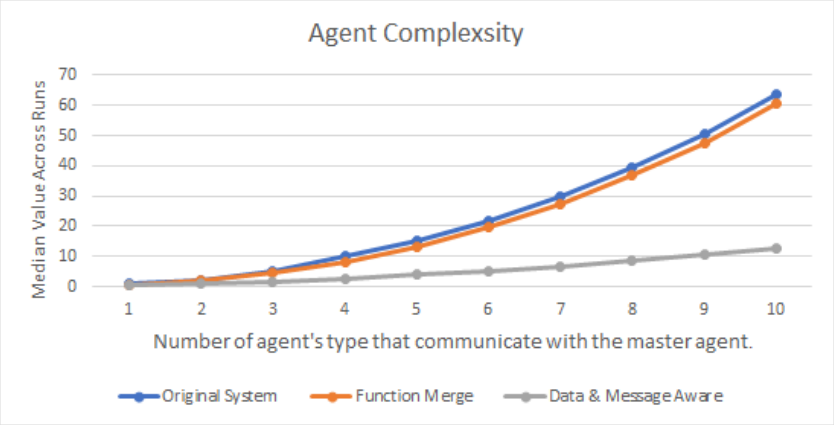

Figure 9.2: Comparison of the median value of processing time against the number of communicating agents (slave-to-master), showing an unmodified system (blue), a system modified to use the functional approach (orange) and a modified system using the data-aware approach (grey).

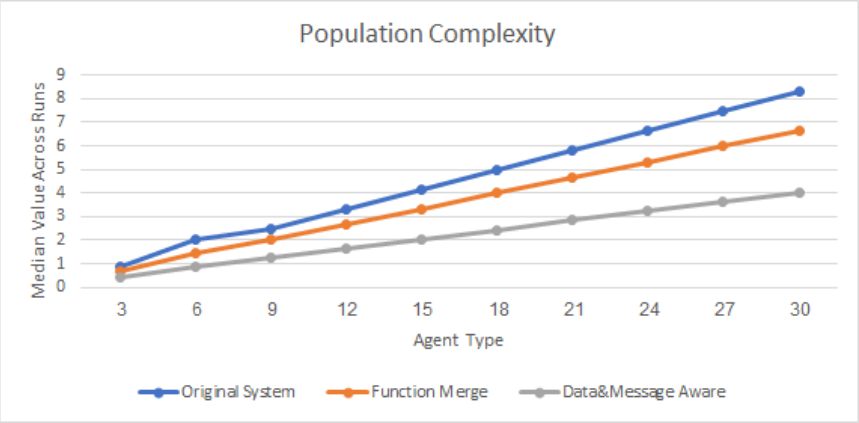

Figure 9.3: Comparison of the median value of execution time against the population divergence, showing an unmodified system (blue), a modified system using the functional approach (orange) and a system modified to use the data-aware approach (grey).

9.2 Limitations of the Research

All the work described in this thesis was demonstrated and tested using FLAME GPU as an ABM software example. Despite this, it is important to consider the limitations of each tool and approach proposed in this work. The major limitation is that both the benchmark and performance enhancement approaches were tested using FLAME GPU and were not tested on any other simulation tool. The functional approach specifically for FLAME GPU, targeting the function layers, reduced the limited amount of agent memory, and it had no effect on reducing message memory access time. The next section highlights a number of topics to be a baseline for further work in this area of research.

9.3 Future Work

The following points highlight potential areas for future research:

- The benchmark model was implemented for the FLAME GPU framework. The ABM community will benefit from replicating this model in a suitable way for other ABM platforms. To formalise and re-implement this model, we aim to use the CoSMoS approach to make it more understandable for other users. The complex system modelling and simulation (CoSMoS) approach [10] was developed to be used as a scientific instrument for understanding a complex system. It offers a guide to modelling and simulating complex systems and integrates all-round verification and validation. The CoSMoS approach has already been successfully used in a number of studies of biological systems, such as [5, 47, 18]. Despite this specific use, CoSMoS can also be tailored and adapted for other uses, according to [38].

- The great achievements of the data-aware approach make it a remarkable choice for testing on other systems. Moreover, the underlying model abstraction is appropriate for any streaming-based MAS platform or model.

Appendices

Appendix A

Functions.c File

```
/*
*
* Functions.c for The Benchmark Model.
*
* Author: Eidah Alzahrani
*
*/
#include <header.h>
#include <vector>

#ifndef _FLAMEGPU_FUNCTIONS
#define _FLAMEGPU_FUNCTIONS
#define AGENT_STATE_DEAD 3
#define AGENT_STATE_BIND 2
#define XMAX    10.0f
#define YMAX        10.0f
#define ZMAX        10.0f
#define radius 1.0f//Interaction radius
#define DT 0.01
#define MOVEMENT_TIME_RANGE 15//by increasing this we will make
    agents move across a larger amount of area and increase the
    chance that they will be within range of another agent to
    interact with
#define MIN_MOVEMENT_TIME 5

std::vector<int>Acounter;
std::vector<int>Bcounter;
std::vector<int>Ccounter;
std::vector<int>iteration;
unsigned int h_iteration = 0;
```

```
28 #define AGENT_STATE_A_DEFAULT 1
29 #define AGENT_STATE_B_DEFAULT 4
30 #define AGENT_STATE_C_DEFAULT 5
31 /*
32  *init_Function.....
33 */
34 __FLAME_GPU_INIT_FUNC__ void initFunction(){
35
36 Acounter.push_back(get_agent_A_moving_A_count());
37 Bcounter.push_back(get_agent_B_moving_B_count());
38 Ccounter.push_back(get_agent_C_moving_C_count());
39 iteration.push_back(0);
40 fflush(stdout);
41  }
42 /*
43  *step_Function.....
44 */
45 __FLAME_GPU_STEP_FUNC__ void stepFunction(){
46
47 h_iteration++;
48 iteration.push_back(h_iteration);
49 Acounter.push_back(get_agent_A_moving_A_count());
50 Bcounter.push_back(get_agent_B_moving_B_count());
51 Ccounter.push_back(get_agent_C_moving_C_count());
52 fflush(stdout);
53 }
54 /*
55  *Exit_Function.....
56 */
57 __FLAME_GPU_EXIT_FUNC__ void exitFunction(){
58
59 FILE *output = fopen("output.dat",  "w");
60
61 fprintf(output, "#IA B C \n ");
62
63 for (int i = 0; i < h_iteration; i++){
64 fprintf(output, "%u %d %d %d \n ", iteration[i],Acounter[i],
      Bcounter[i], Ccounter[i]);
65 }
66 fclose(output);
67 }
68 /*
```

```
 * move_A FLAMEGPU Agent Function
*/
__FLAME_GPU_FUNC__ int move_A(xmachine_memory_A* agent, RNG_rand48*
    rand48){

  float vx;
  float vy;
  float vz;

  float x = agent->x;
  float y = agent->y;
  float z = agent->z;

  //generate a new direction by creating a new random velocity
   if (agent->count == 0) {
      float r_x = (rnd(rand48) - 0.5f)*2.0f;
     float r_y = (rnd(rand48) - 0.5f)*2.0f;
     float r_z = (rnd(rand48) - 0.5f)*2.0f;

      agent->vx = r_x;
      agent->vy = r_y;
      agent->vz = r_z;

      agent->count = (int)(rnd(rand48)*(float)MOVEMENT_TIME_RANGE)
   + MIN_MOVEMENT_TIME;
 }
  //get the velocity
      vx=agent->vx ;
      vy=agent->vy ;
      vz=agent->vz ;

  //move according to velocity
      x = x + vx*DT;
      y = y + vy*DT;
      z = z + vz*DT;

//(Clamp position to environment)
   x = x >= XMAX ? XMAX : x;
   x = x <= 0.0 ? 0.0 : x;
   y = y >= YMAX ? YMAX : y;
   y = y <= 0.0 ? 0.0 : y;
   z = z >= ZMAX ? ZMAX : z;
```

```
109    z = z <= 0.0 ? 0.0 : z;
110
111     agent->x = x;
112        agent->y = y;
113        agent->z = z;
114        agent->count--;
115        agent->state = AGENT_STATE_A_DEFAULT;
116
117 return 0;
118 }
119 /*
120  * death_A FLAMEGPU Agent Function
121 */
122 __FLAME_GPU_FUNC__ int death_A(xmachine_memory_A* agent){
123
124   return 1;
125
126 }
127 /*
128  * move_B FLAMEGPU Agent Function
129 */
130 __FLAME_GPU_FUNC__ int move_B(xmachine_memory_B* agent, RNG_rand48*
      rand48){
131
132   float vx;
133   float vy;
134   float vz;
135
136   float x = agent->x;
137   float y = agent->y;
138   float z = agent->z;
139
140   //generate a new direction by creating a new random velocity
141    if (agent->count == 0) {
142       float r_x = (rnd(rand48) - 0.5f)*2.0f;
143      float r_y = (rnd(rand48) - 0.5f)*2.0f;
144      float r_z = (rnd(rand48) - 0.5f)*2.0f;
145
146       agent->vx = r_x;
147       agent->vy = r_y;
148       agent->vz = r_z;
149
```

```
150       agent->count = (int)(rnd(rand48)*(float)MOVEMENT_TIME_RANGE)
    + MIN_MOVEMENT_TIME;
151  }
152   //get the velocity
153       vx=agent->vx ;
154       vy=agent->vy ;
155       vz=agent->vz ;
156
157   //move according to velocity
158       x = x + vx*DT;
159       y = y + vy*DT;
160       z = z + vz*DT;
161
162 //(Clamp position to environment)
163    x = x >= XMAX ? XMAX : x;
164    x = x <= 0.0 ? 0.0 : x;
165    y = y >= YMAX ? YMAX : y;
166    y = y <= 0.0 ? 0.0 : y;
167    z = z >= ZMAX ? ZMAX : z;
168    z = z <= 0.0 ? 0.0 : z;
169
170     agent->x = x;
171        agent->y = y;
172        agent->z = z;
173        agent->count--;
174        agent->state = AGENT_STATE_B_DEFAULT;
175
176 return 0;
177 }
178 /*
179  * send_locationB FLAMEGPU Agent Function
180 */
181 __FLAME_GPU_FUNC__ int send_locationB(xmachine_memory_B* agent,
    xmachine_message_locationB_list* locationB_messages){
182
183     int id, s, t, c_id;
184     float x, y, z, c_point;
185     id = agent->id;
186     x = agent->x;
187    y = agent->y;
188     z = agent->z;
189     s = agent->state;
```

```
190    t = agent->type;
191   c_point = agent->closest_point;
192 c_id = agent->closest_id;
193 add_locationB_message(locationB_messages, c_id ,c_point , id, s,t,
      x, y, z );
194
195 return 0;
196 }
197
198 /*
199  * receive_bindB FLAMEGPU Agent Function
200 */
201 __FLAME_GPU_FUNC__ int receive_bindB(xmachine_memory_B* agent,
      xmachine_message_bindB_list* bindB_messages,
      xmachine_message_bindB_PBM* partition_matrix){
202    int c = 0, nearest_id = 0;
203   float nearest_distance = 0.0f;
204  xmachine_message_bindB* current_message = get_first_bindB_message(
     bindB_messages, partition_matrix, agent->x, agent->y, agent->z);
205
206   while (current_message)
207    {
208      if (current_message->id != agent->id){
209         if (agent->id == current_message->closest_id){
210            if (c == 0){
211                c++;
212                nearest_distance = current_message->closest_point;
213                nearest_id = current_message->id;
214              }
215              else if (nearest_distance > current_message->
     closest_point){
216                  nearest_distance = current_message->closest_point;
217                  nearest_id = current_message->id;
218              }
219          }
220     }
221    current_message = get_next_bindB_message(current_message,
     bindB_messages, partition_matrix);
222
223    }
224  if (c == 1) {
225 agent->state = AGENT_STATE_DEAD;
```

```
226 agent->closest_point = nearest_distance;
227 agent->closest_id = nearest_id;
228 }
229   else{
230      agent->closest_id = -1;
231    }
232 
233 return 0;
234 }
235 /*
236  * send_combinedB FLAMEGPU Agent Function
237 */
238 __FLAME_GPU_FUNC__ int send_combinedB(xmachine_memory_B*  agent ,
      xmachine_message_combinedB_list* combinedB_messages){
239 
240 int id, s, t, c_id;
241 float x, y, z, c_point;
242 id = agent->id;
243 x = agent->x;
244 y = agent->y;
245 z = agent->z;
246 s = agent->state;
247 t = agent->type;
248 c_point = agent->closest_point;
249 c_id = agent->closest_id;
250 add_combinedB_message(combinedB_messages, c_id ,c_point , id, s,t,
      x, y, z );
251 return 1;
252 }
253 /*
254  * need_locationB FLAMEGPU Agent Function
255 */
256 __FLAME_GPU_FUNC__ int need_locationB(xmachine_memory_A* agent,
      xmachine_message_locationB_list* locationB_messages,
      xmachine_message_locationB_PBM* partition_matrix){
257 
258     int c = 0, nearest_id;
259     float distance_check, x1, x2, y1, y2, z1, z2;
260     float nearest_distance = 0.0f;
261     x1 = agent->x;
262    y1 = agent->y;
263     z1 = agent->z;
```

```
264
265 xmachine_message_locationB* current_message =
    get_first_locationB_message(locationB_messages, partition_matrix
    , agent->x, agent->y, agent->z);
266
267   while (current_message)
268    {
269       if (current_message->id != agent->id){
270            x2 = current_message->x;
271           y2 = current_message->y;
272           z2 = current_message->z;
273            distance_check = sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(
    y1 - y2) + (z1 - z2)*(z1 - z2));
274                if (distance_check <= radius) {
275                      if (c == 0) {
276                            c++;
277                            nearest_distance = distance_check;
278                            nearest_id = current_message->id;
279                       }
280                  else if (nearest_distance > distance_check){
281                          nearest_distance = distance_check;
282                          nearest_id = current_message->id;
283                      }
284                   }
285           }
286    current_message = get_next_locationB_message(current_message,
    locationB_messages, partition_matrix);
287
288    }
289  if (c == 1) {
290 agent->state = AGENT_STATE_BIND;
291 agent->closest_point = nearest_distance;
292 agent->closest_id = nearest_id;
293 }
294   else{
295      agent->closest_id = -1;
296    }
297
298 return 0;
299 }
300 /*
301  * send_bindB FLAMEGPU Agent Function
```

```
302 */
303 __FLAME_GPU_FUNC__ int send_bindB(xmachine_memory_A* agent,
        xmachine_message_bindB_list* bindB_messages){
304
305 int id, s, t, c_id;
306   float x, y, z, c_point;
307 id = agent->id;
308   x = agent->x;
309 y = agent->y;
310 z = agent->z;
311 t = agent->type;
312   s = agent->state;
313   c_point = agent->closest_point;
314 c_id = agent->closest_id;
315 add_bindB_message(bindB_messages, c_id ,c_point , id, s,t, x, y, z
        );
316 return 0;
317 }
318 /*
319  * created_C FLAMEGPU Agent Function
320 */
321 __FLAME_GPU_FUNC__ int created_C0(xmachine_memory_A* agent,
        xmachine_memory_C_list* C_agents,
        xmachine_message_combinedB_list* combinedB_messages,
        xmachine_message_combinedB_PBM* partition_matrix, RNG_rand48*
        rand48){
322 int c = 0;
323
324   xmachine_message_combinedB* current_message =
        get_first_combinedB_message(combinedB_messages, partition_matrix
        , agent->x, agent->y, agent->z);
325
326   while (current_message)
327   {
328       if (current_message->id != agent->id){
329       if (current_message->closest_id == agent->id) {
330 c++;
331      }
332     else
333     {
334         current_message->state = AGENT_STATE_DEAD;
335     }
```

```
336     }
337       current_message = get_next_combinedB_message(current_message,
        combinedB_messages, partition_matrix);
338          }
339          if(c >= 1) {
340         agent->state = 6;
341         add_C_agent(C_agents, 0.0, 0, agent->count, agent->id,
       AGENT_STATE_C_DEFAULT, 1 ,agent->vx, agent->vy, agent->vz, agent
       ->x, agent->y, agent->z );
342     }
343      return 0;
344   }
345 /*
346  * move_C FLAMEGPU Agent Function
347 */
348 __FLAME_GPU_FUNC__ int move_C(xmachine_memory_C* agent, RNG_rand48*
        rand48){
349 
350   float vx;
351   float vy;
352   float vz;
353 
354   float x = agent->x;
355   float y = agent->y;
356   float z = agent->z;
357 
358   //generate a new direction by creating a new random velocity
359    if (agent->count == 0) {
360       float r_x = (rnd(rand48) - 0.5f)*2.0f;
361      float r_y = (rnd(rand48) - 0.5f)*2.0f;
362      float r_z = (rnd(rand48) - 0.5f)*2.0f;
363 
364       agent->vx = r_x;
365       agent->vy = r_y;
366       agent->vz = r_z;
367 
368       agent->count = (int)(rnd(rand48)*(float)MOVEMENT_TIME_RANGE)
       + MIN_MOVEMENT_TIME;
369  }
370   //get the velocity
371       vx=agent->vx ;
372       vy=agent->vy ;
```

```
373       vz=agent->vz ;
374
375   //move according to velocity
376       x = x + vx*DT;
377       y = y + vy*DT;
378       z = z + vz*DT;
379
380 //(Clamp position to environment)
381    x = x >= XMAX ? XMAX : x;
382    x = x <= 0.0 ? 0.0 : x;
383    y = y >= YMAX ? YMAX : y;
384    y = y <= 0.0 ? 0.0 : y;
385    z = z >= ZMAX ? ZMAX : z;
386    z = z <= 0.0 ? 0.0 : z;
387
388     agent->x = x;
389        agent->y = y;
390        agent->z = z;
391        agent->count--;
392        agent->state = AGENT_STATE_C_DEFAULT;
393
394 return 0;
395 }
396 #endif //_FLAMEGPU_FUNCTIONS
```

Listing A.1: Functions.c of the model that represents A+B

Appendix B

Scanner.l File

```
1
2  %{
3  //some C declarations
4  %}
5
6  //Flex Definition section
7  %option noyywrap
8  %option yylineno
9
10 O   [0-7]
11 D   [0-9]
12 NZ  [1-9]
13 L   [a-zA-Z_]
14 A   [a-zA-Z_0-9]
15 H   [a-fA-F0-9]
16 HP  (0[xX])
17 E   ([Ee][+-]?{D}+)
18 P   ([Pp][+-]?{D}+)
19 FS  (f|F|l|L)
20 IS  (((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?))
21 CP  (u|U|L)
22 SP  (u8|u|U|L)
23 WS  [ \t\v\n\f]
24 white_space   [ \t\n]
25 blank     [ \t]
26 other     .
27
28 %%
29 "alignof"          return ALIGNOF ;
30 "and"              return AND ;
```

```
31 "and_eq"           return and_EG ;
32 "asm"              return ASM ;
33 "auto"             return AUTO ;
34 "bitand"           return BITAND ;
35 "bitor"            return BITOR ;
36 "bool"             return BOOL ;
37 "break"            return BREAK ;
38 "case"             return CASE ;
39 "catch"            return CATCH ;
40 "char"             return CHAR ;
41 "char16_t"         return CHAR16_T ;
42 "char32_t"         return CHAR32_T ;
43 "cin"              return CIN ;
44 "class"            return CLASS ;
45 "compl"            return COMPLE ;
46 "const"            return CONST ;
47 "constexpr"        return CONSTEXPR ;
48 "const_cast"       return CONST_CAST;
49 "continue"         return CONTINUE ;
50 "cout"             return COUT ;
51 "decltype"         return DECLTYPE ;
52 "default"    { return(DEFAULT); }
53 "delete"           return DELETE ;
54 "do"               return DO;
55 "double"           return DOUBLE ;
56 "dynamic_cast"     return DYNAMIC_CAST;
57 "else"             return ELSE ;
58 "endl"             return ENDL ;
59 "enum"             return ENUM ;
60 "explicit"         return EXPLICIT ;
61 "export"           return EXPORT ;
62 "extern"           return EXTERN ;
63 "false"            return FALSE ;
64 "float"            return FLOAT ;
65 "for"              { return FOR ;}
66 "friend"           return FRIEND ;
67 "goto"             return GOTO ;
68 "if"               return IF ;
69 "include"          return INCLUDE ;
70 "inline"           return INLINE ;
71 "int"            {  return INT ;}
72 "INT_MAX"          return INT_MAX ;
```

```
73  "INT_MIN"           return INT_MIN ;
74  "iomanip"           return IOMANIP ;
75  "iostream"          return IOSTREAM ;
76  "long"              return LONG ;
77  "main "             return MAIN ;
78  "MAX_RAND"          return MAX_RAND ;
79  "mutable"           return MUTABLE ;
80  "namespace"         return NAMESPACE ;
81  "new"               return NEW ;
82  "noexcept"          return NOEXCEPT ;
83  "not"               return NOT ;
84  "not_eq"            return NOT_EG ;
85  "npos"              return NPOS ;
86  "nullptr"           return NULLPTR ;
87  "operator"          return OPERATOR ;
88  "or"                return OR ;
89  "or_eq"             return OR_EQ;
90  "private"           return PRIVATE ;
91  "protected"         return PRPTECTED ;
92  "public"            return PUBLIC ;
93  "register"          return REGISTER ;
94  "restrict"        return(RESTRICT);
95  "reinterpret_cast" return REINTERPRET_CAST ;
96  "return"            return RETURN ;
97  "short"             return SHORT ;
98  "signed"            return SIGNED ;
99  "sizeof"            return SIZEOF ;
100 "static"            return STATIC ;
101 "static_assert"     return STATIC_ASSERT;
102 "static_cast"       return STATIC_CAST ;
103
104 "string"            return STRING ;
105 "struct"            return STRUCT ;
106 "switch"            return SWITCH ;
107 "template"          return TEMPLATE;
108 "this"              return THIS ;
109 "thread_local"      return THREAD_LOCAL ;
110 "throw"             return THROW ;
111 "true"              return TRUE ;
112 "try"               return TRY ;
113 "typedef"           return TYPEDEF ;
114 "typeid"            return TYPEID ;
```

```
115 "typename"          return TYPENAME ;
116 "union"             return UNION ;
117 "unsigned"          return UNSIGNED ;
118 "using"             return USING ;
119 "virtual"           return VIRTUAL ;
120 "void"              return VOID ;
121 "volatile"          return VOLATILE ;
122 "wchar_t"           return WCHAR_T ;
123 "while"             return WHILE ;
124 "xor"               return XOR ;
125 "xor_eq"            return XOR_EG ;
126 "_Alignas"          return ALIGNAS;
127 "_Alignof"          return ALIGNOF;
128 "_Atomic"           return ATOMIC;
129 "_Generic"          return GENERIC;
130 "_Imaginary"        return IMAGINARY;
131 "_Noreturn"         return NORETURN;
132 "__func__"          return FUNC_NAME;
133
134 [L]([L]|[D])*    {return(check_type()); }
135 {L}{A}*          { return check_type(); }
136
137 {HP}{H}+{IS}?        {  return I_CONSTANT; }
138 {NZ}{D}*{IS}?        {  return I_CONSTANT; }
139 "0"{O}*{IS}?         {  return I_CONSTANT; }
140 {D}+{E}{FS}?         {  return F_CONSTANT; }
141 {D}*"."{D}+{E}?{FS}?     {  return F_CONSTANT; }
142 {D}+"."{E}?{FS}?       {  return F_CONSTANT; }
143 {HP}{H}+{P}{FS}?       {  return F_CONSTANT; }
144 {HP}{H}*"."{H}+{P}{FS}?    {  return F_CONSTANT; }
145 {HP}{H}+"."{P}{FS}?      {  return F_CONSTANT; }
146
147 ({SP}?\"([^"\\\n]|{ES})*\"{WS}*)+ {    return STRING_LITERAL; }
148
149 "..."        return(ELLIPSIS);
150 ">>="        return(RIGHT_ASSIGN);
151 "<<="        return(LEFT_ASSIGN);
152 "+="         return(ADD_ASSIGN);
153 "-="           return(SUB_ASSIGN);
154 "*="         return(MUL_ASSIGN);
155 "/="         return(DIV_ASSIGN);
156 "%="         return(MOD_ASSIGN);
```

```
"&="          return(AND_ASSIGN);
"^="          return(XOR_ASSIGN);
"|="          return(OR_ASSIGN);
">>"          return(RIGHT_OP);
"<<"          return(LEFT_OP);
"++"          return(INC_OP);
"--"          return(DEC_OP);
"->"            return(PTR_OP);
"&&"          return(AND_OP);
"||"          return(OR_OP);
"<="          return(LE_OP);
">="          return(GE_OP);
"=="          return(EQ_OP);
"!="          return(NE_OP);
";"             return(';');
("{"|"<%")        return('{');
("}"|"%>")          return('}');
","             return(',');
":"             return(':');
"="             return('=');
"("             return('(');
")"             return(')');
("["|"<:")        return('[');
("]"|":>")        return(']');
"."             return('.');
"&"           return('&');
"!"             return('!');
"~"             return('~');
"-"             return('-');
"+"             return('+');
"*"             return ('*');
"/"             return('/');
"%"             return('%');
"<"             return('<');
">"             return('>');
"^"             return('^');
"|"             return('|');
"?"             return('?');
"#endif"         return ENDIF;
"#ifndef"            return IFNDEF;
"#define"            return DEFINE;
"\n" { yylineno= lineno++; };
```

```
199 %%
200
201 void parseFile( FILE *file ) {
202
203   yyin = file;
204 }
```

Listing B.1: Functions.c of the model that represents A+B

Appendix C

Meta-data.xslt File

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xmml="http://www.dcs.shef.ac.uk/~paul/XMML"
                xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
        exclude-result-prefixes="xmml gpu xsl " >


   <xsl:output indent="yes" method="xml" encoding="utf-8" omit-xml-declaration="yes"/>
 <xsl:strip-space elements="*"/>
<xsl:variable name="lookup-source" select="document('output1.xml')" />
<xsl:key name="dependency" match="data_dependency" use="functions/gpu:function/name" />


<xsl:template match="gpu:function" >
 <gpu:function>
    <xsl:variable name="function_name" select="xmml:name"/>

    <!-- Output ANY elements which may occur before our new dependency elements -->

    <xsl:apply-templates select="xmml:name"/>
    <xsl:apply-templates select="xmml:currentState"/>
    <xsl:apply-templates select="xmml:nextState"/>

    <!-- Output ANY elements for our new in_dependency elements -->
    <xsl:for-each select="$lookup-source/functions/function[name=$function_name]">
```

```
<xsl:if test="In_data='true'">
      <in_datadependency>
      <xsl:for-each select="In_datadependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </in_datadependency>
  </xsl:if>
  </xsl:for-each>

<!-- Output ANY elements for our new out_dependency elements -->
  <xsl:for-each select="$lookup-source/functions/function[name=$
 function_name]">
<xsl:if test="Out_data='true'">
      <out_datadependency>
      <xsl:for-each select="Out_datadependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </out_datadependency>
  </xsl:if>
  </xsl:for-each>

<!-- Output ANY elements for our new out_dependency elements -->
  <xsl:for-each select="$lookup-source/functions/function[name=$
 function_name]">
<xsl:if test="In_message='true'">
      <in_messagedependency>
      <xsl:for-each select="In_messagedependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </in_messagedependency>
  </xsl:if>
  </xsl:for-each>

  <!-- Output ANY elements which may occur after our new
 dependency elements (TODO: All cases of input/outputs/conditions
 / global conditions etc.) -->
```

```
    <xsl:apply-templates select="xmml:outputs"/>
    <xsl:apply-templates select="xmml:inputs"/>
  <xsl:apply-templates select="xmml:xagentOutputs"/>
  <xsl:apply-templates select="gpu:globalCondition"/>
  <xsl:apply-templates select="xmml:condition"/>
    <xsl:apply-templates select="gpu:reallocate"/>
    <xsl:apply-templates select="gpu:RNG"/>
  </gpu:function>
</xsl:template>



<!-- identity transform -->
<xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>


</xsl:stylesheet>
```

Listing C.1: Functions.c of the model that represents A+B

Appendix D

Meta-data.xslt File

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xmml="http://www.dcs.shef.ac.uk/~paul/XMML"
                xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
        exclude-result-prefixes="xmml gpu xsl " >


   <xsl:output indent="yes" method="xml" encoding="utf-8" omit-xml-declaration="yes"/>
 <xsl:strip-space elements="*"/>
<xsl:variable name="lookup-source" select="document('output1.xml')" />
<xsl:key name="dependency" match="data_dependency" use="functions/gpu:function/name" />


<xsl:template match="gpu:function" >
 <gpu:function>
    <xsl:variable name="function_name" select="xmml:name"/>

    <!-- Output ANY elements which may occur before our new dependency elements -->

    <xsl:apply-templates select="xmml:name"/>
    <xsl:apply-templates select="xmml:currentState"/>
    <xsl:apply-templates select="xmml:nextState"/>

    <!-- Output ANY elements for our new in_dependency elements -->
    <xsl:for-each select="$lookup-source/functions/function[name=$function_name]">
```

```
<xsl:if test="In_data='true'">
      <in_datadependency>
      <xsl:for-each select="In_datadependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </in_datadependency>
  </xsl:if>
  </xsl:for-each>

<!-- Output ANY elements for our new out_dependency elements -->
  <xsl:for-each select="$lookup-source/functions/function[name=$
 function_name]">
<xsl:if test="Out_data='true'">
      <out_datadependency>
      <xsl:for-each select="Out_datadependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </out_datadependency>
  </xsl:if>
  </xsl:for-each>

<!-- Output ANY elements for our new out_dependency elements -->
  <xsl:for-each select="$lookup-source/functions/function[name=$
 function_name]">
<xsl:if test="In_message='true'">
      <in_messagedependency>
      <xsl:for-each select="In_messagedependency/variable">
          <dependencyVariable><name>
    <xsl:value-of select="name"/>
    </name></dependencyVariable>
      </xsl:for-each>
      </in_messagedependency>
  </xsl:if>
  </xsl:for-each>

  <!-- Output ANY elements which may occur after our new
 dependency elements (TODO: All cases of input/outputs/conditions
 / global conditions etc.) -->
```

```
    <xsl:apply-templates select="xmml:outputs"/>
    <xsl:apply-templates select="xmml:inputs"/>
  <xsl:apply-templates select="xmml:xagentOutputs"/>
  <xsl:apply-templates select="gpu:globalCondition"/>
  <xsl:apply-templates select="xmml:condition"/>
    <xsl:apply-templates select="gpu:reallocate"/>
    <xsl:apply-templates select="gpu:RNG"/>
  </gpu:function>
</xsl:template>



<!-- identity transform -->
<xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>


</xsl:stylesheet>
```

Listing D.1: Functions.c of the model that represents A+B

Bibliography

[1] Brandon G Aaby, Kalyan S Perumalla, and Sudip K Seal. "Efficient simulation of agent-based models on multi-GPU and multi-core clusters". In: *Proceedings of the 3rd international ICST conference on simulation tools and techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2010, p. 29.

[2] Sameera Abar et al. "Agent Based Modelling and Simulation tools: A review of the state-of-art software". In: *Computer Science Review* 24 (2017), pp. 13–33.

[3] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[4] Samuel Alberts et al. "Data-parallel techniques for simulating a mega-scale agent-based model of systemic inflammatory response syndrome on graphics processing units". In: *Simulation* 88.8 (2012), pp. 895–907.

[5] Kieran Alden et al. "Pairing experimentation and computational modeling to understand the role of tissue inducer cells in the development of lymphoid organs". In: *Frontiers in Immunology* 3 (2012), p. 172.

[6] Kieran Alden et al. "Utilising a simulation platform to understand the effect of domain model assumptions". In: *Natural computing* 14.1 (2015), pp. 99–107.

[7] Robert John Allan. *Survey of agent based modelling and simulation tools*. Science & Technology Facilities Council, 2010.

[8] Eidah Alzahrani, Paul Richmond, and Anthony JH Simons. "A formula-driven scalable benchmark model for ABM, applied to FLAME GPU". In: *European Conference on Parallel Processing*. Springer. 2017, pp. 703–714.

[9] Eidah Alzahrani, Anthony JH Simons, and Paul Richmond. "Data Aware Simulation of Complex Systems on GPUs". In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 567–574.

[10] Paul S Andrews et al. "CoSMoS process, models, and metamodels". In: *Proceedings of the 2011 Workshop on Complex Systems Modelling and Simulation, Paris, France*. 2011, pp. 1–13.

[11] Kishoj Bajracharya and Raphael Duboz. "Comparison of three agent-based platforms on the basis of a simple epidemiological model (WIP)". In: *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*. Society for Computer Simulation International. 2013, p. 7.

[12] José Barbosa and Paulo Leitão. "Simulation of multi-agent manufacturing systems using agent-based modelling platforms". In: *2011 9th IEEE International Conference on Industrial Informatics*. IEEE. 2011, pp. 477–482.

[13] Isaac Sánchez Barrera et al. "Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies". In: (2018).

[14] Anthony Bigbee, Claudio Cioffi-Revilla, and Sean Luke. "Replication of Sugarscape using MASON". In: *Agent-Based Approaches in Economic and Social Complex Systems IV*. Springer, 2007, pp. 183–190.

[15] William Blume and Rudolf Eigenmann. "Nonlinear and symbolic data dependence testing". In: *IEEE Transactions on Parallel and Distributed Systems* 9.12 (1998), pp. 1180–1194.

[16] Andrei Borshchev and Alexei Filippov. "From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools". In: *Proceedings of the 22nd International Conference of the System Dynamics Society*. Vol. 22. Citeseer. 2004.

[17] LB Bosi, M Mariotti, and A Santocchia. "GPU Linear algebra extensions for GNU/Octave". In: *Journal of Physics: Conference Series*. Vol. 368. 1. IOP Publishing. 2012, p. 012062.

[18] James Bown et al. "Engineering simulations for cancer systems biology". In: *Current Drug Targets* 13.12 (2012), pp. 1560–1574.

[19] Federico Campeotto, Agostino Dovier, and Enrico Pontelli. "Protein structure prediction on GPU: a declarative approach in a multi-agent framework". In: *2013 42nd International Conference on Parallel Processing*. IEEE. 2013, pp. 474–479.

[20] José M Cecilia et al. "Enhancing data parallelism for ant colony optimization on GPUs". In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 42–51.

[21] Wenan Chen et al. "Agent based modeling of blood coagulation system: implementation using a GPU based high speed framework". In: *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE. 2011, pp. 145–148.

[22] Mozhgan K Chimeh and Paul Richmond. "Simulating heterogeneous behaviours in complex systems on GPUs". In: *Simulation Modelling Practice and Theory* 83 (2018), pp. 3–17.

[23] S. Chin. *FLAME Overview*. 2015. URL: http://flame.ac.uk/docs/overview.html (visited on 09/30/2018).

[24] Timothy Chuang and Munehiro Fukuda. "A parallel multi-agent spatial simulation environment for cluster systems". In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. IEEE. 2013, pp. 143–150.

[25] Simon Coakley, Rod Smallwood, and Mike Holcombe. "Using x-machines as a formal basis for describing agents in agent-based modelling". In: *Simulation Series* 38.2 (2006), p. 33.

[26] Simon Coakley et al. "Exploitation of high performance computing in the FLAME agent-based simulation framework". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE. 2012, pp. 538–545.

[27] Nicholson Collier and Michael North. "Parallel agent-based simulation with repast for high performance computing". In: *Simulation* 89.10 (2013), pp. 1215–1235.

[28] Nicholson Collier and Michael North. "Repast HPC: A platform for large-scale agent-based modeling". In: *Large-Scale Computing* 10 (2012), pp. 81–109.

[29] Nick Collier. "Repast: An extensible framework for agent simulation". In: *The University of Chicago's Social Science Research* 36 (2003), p. 2003.

[30] Gennaro Cordasco et al. "Bringing together efficiency and effectiveness in distributed simulations: the experience with D-MASON". In: *Simulation* 89.10 (2013), pp. 1236–1253.

[31] Gennaro Cordasco et al. "D-Mason: A Distributed Framework for Agent Based Simulations". In: *Submitted for Publication* (2012).

[32] Anthony J Cowling et al. "Communicating stream X-machines systems are no more than X-machines". In: *Journal of Universal Computer Science* 5.9 (1999), pp. 494–507.

[33] Laurence Dawson and Iain Stewart. "Improving Ant Colony Optimization performance on the GPU using CUDA". In: *2013 IEEE Congress on Evolutionary Computation*. IEEE. 2013, pp. 1901–1908.

[34] Christophe Deissenberg, Sander Van Der Hoog, and Herbert Dawid. "EURACE: A massively parallel agent-based model of the European economy". In: *Applied Mathematics and Computation* 204.2 (2008), pp. 541–552.

[35] Audrey DeléVacq et al. "Parallel ant colony optimization on graphics processing units". In: *Journal of Parallel and Distributed Computing* 73.1 (2013), pp. 52–61.

[36] Lorenzo Dematte. "Parallel particle-based reaction diffusion: a GPU implementation". In: *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. IEEE. 2010, pp. 67–77.

[37] Lorenzo Dematté and Davide Prandi. "GPU computing for systems biology". In: *Briefings in Bioinformatics* 11.3 (2010), pp. 323–333.

[38] Ali Afshar Dodson et al. "Using the CoSMoS approach to study Schelling's bounded neighbourhood model". In: *CoSMoS 2014* (2014), p. 1.

[39] Samuel Eilenberg. *Automata, languages, and machines*. Academic press, 1974.

[40] Joshua M Epstein and Robert Axtell. *Growing artificial societies: social science from the bottom up*. Brookings Institution Press, 1996.

[41] Ugo Erra et al. "Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance". In: *Proceedings of Vision, Modeling and Visualization 2004 (VMV)* (2004).

[42] Nuno Fachada et al. "Towards a standard model for research in agent-based modeling and simulation". In: *PeerJ Computer Science* 1 (2015), e36.

[43] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *Parallel Processing (ICPP), 2011 International Conference on*. IEEE. 2011, pp. 216–225.

[44] Grazziela P Figueredo, Uwe Aickelin, and Peer-Olaf Siebers. "Systems dynamics or agent-based modelling for immune simulation?" In: *International Conference on Artificial Immune Systems*. Springer. 2011, pp. 81–94.

[45] Munehiro Fukuda. "Mass: Parallel-computing library for multi-agent spatial simulation". In: *Distributed Systems Laboratory, Computing & Software Systems, University of Washington Bothell, Bothell, WA, http://depts. washington. edu/dslab/SensorGrid/doc/MassSpec. pdf* (2010).

[46] Michael Garland et al. "Parallel computing experiences with CUDA". In: *IEEE micro* 4 (2008), pp. 13–27.

[47] Philip Garnett et al. "Using the CoSMoS process to enhance an executable model of auxin transport canalisation". In: *CoSMoS* 2010 (2010), pp. 9–32.

[48] B DANTZIG GEORGE. "FOURIER-MOTZKIN ELIMINATION AND ITS DUAL"". In: *The Basic George B. Dantzig* (2003), p. 255.

[49] Michael E Goldsby and Carmen M Pancerella. "Multithreaded agent-based simulation". In: *Simulation Conference (WSC), 2013 Winter*. IEEE. 2013, pp. 1581–1591.

[50] Poonam Goyal et al. “A fast, scalable SLINK algorithm for commodity cluster computing exploiting spatial locality”. In: *2016 IEEE 18th International Conference on High-Performance Computing and Communications, IEEE 14th International Conference on Smart City, and IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2016, pp. 268–275.

[51] Volker Grimm and Steven F Railsback. “Agent-based models in ecology: patterns and alternative theories of adaptive behaviour”. In: *Agent-based computational modelling*. Springer, 2006, pp. 139–152.

[52] Volker Grimm et al. “Pattern-oriented modeling of agent-based complex systems: lessons from ecology”. In: *science* 310.5750 (2005), pp. 987–991.

[53] Mark Harris, Shubhabrata Sengupta, and John D Owens. “Parallel prefix sum (scan) with CUDA”. In: *GPU gems* 3.39 (2007), pp. 851–876.

[54] Nathaniel Breault Hart. “MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks”. PhD thesis. 2015.

[55] Blake Haugen et al. “Visualizing execution traces with task dependencies”. In: *Proceedings of the 2nd Workshop on Visual Performance Analysis*. ACM. 2015, p. 2.

[56] Emmanuel Hermellin and Fabien Michel. “GPU delegation: Toward a generic approach for developping MABS using GPU programming”. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2016, pp. 1249–1258.

[57] Emmanuel Hermellin and Fabien Michel. “Overview of case studies on adapting MABS models to GPU programming”. In: *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer. 2016, pp. 125–136.

[58] Jared Hoberock and Nathan Bell. *Thrust: A parallel template library*. 2010.

[59] Hitoshi Iba. *Agent-based Modeling and Simulation with Swarm*. Chapman and Hall/CRC, 2013.

[60] Rafia Inam. *An introduction to gpgpu programming-cuda architecture*. Mälardalen University, Mälardalen Real-Time Research Centre, 2010.

[61] Tim Ingham-Dempster, Bernard Corfe, and Dawn Walker. "A cellular based model of the colon crypt suggests novel effects for Apc phenotype in colorectal carcinogenesis". In: *Journal of Computational Science* 24 (2018), pp. 125–131.

[62] Tim Ingham-Dempster, Dawn C Walker, and Bernard M Corfe. "An agent-based model of anoikis in the colon crypt displays novel emergent behaviour consistent with biological observations". In: *Royal Society Open Science* 4.4 (2017), p. 160858.

[63] Vandana Jagtap and Urmila Shrawankar. "Dependency analysis for secured code level parallelization". In: *Procedia Computer Science* 78 (2016), pp. 831–837.

[64] Masoud Jalayer, Carlotta Orsenigo, and Carlo Vercellis. "CoV-ABM: A stochastic discrete-event agent-based framework to simulate spatiotemporal dynamics of COVID-19". In: *ArXiv preprint arXiv:2007.13231* (2020).

[65] Kamran Karimi, Neil G Dickson, and Firas Hamze. "A performance comparison of CUDA and OpenCL". In: *arXiv preprint arXiv:1005.2581* (2010).

[66] Rajwinder Kaur and Pawan Luthra. "Load balancing in cloud computing". In: *Proceedings of International Conference on Recent Trends in Information, Telecommunication and Computing, ITC*. Citeseer. 2012.

[67] Foram F Kherani and Jignesh Vania. "Load Balancing in cloud computing". In: (2014).

[68] Mariam Kiran et al. "FLAME: simulating large populations of agents on parallel hardware architectures". In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. 2010, pp. 1633–1636.

[69] DB Kirk and WH Wen-mei. "Programming massively parallel processors: a hands-on approach. Newnes". In: *Google Scholar* (2012).

[70] Dominik Klein, Johannes Marx, and Kai Fischbach. “Agent-Based Modeling in Social Science, History, and Philosophy. An Introduction”. In: *Historical Social Research/Historische Sozialforschung* 43.1 (163 (2018), pp. 7–27.

[71] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. “The I test: an improved dependence test for automatic parallelization and vectorization”. In: *IEEE Transactions on Parallel and Distributed Systems* 2.3 (1991), pp. 342–349.

[72] Elizaveta Kosiachenko. “Efficient GPU Parallelization of the Agent-Based Models Using MASS CUDA Library”. PhD thesis. 2018.

[73] Konstantinos Kyriakopoulos and Kleanthis Psarris. “Data dependence analysis techniques for increased accuracy and extracted parallelism”. In: *International Journal of parallel programming* 32.4 (2004), pp. 317–359.

[74] Guillaume Laville et al. “MCMAS: a toolkit to benefit from many-core architecure in agent-based simulation”. In: *European Conference on Parallel Processing*. Springer. 2013, pp. 544–554.

[75] Guillaume Laville et al. “Using GPU for multi-agent multi-scale simulations”. In: *Distributed Computing and Artificial Intelligence*. Springer, 2012, pp. 197–204.

[76] Bo Li and Ramakrishnan Mukundan. “A comparative analysis of spatial partitioning methods for large-scale, real-time crowd simulation”. In: (2013).

[77] Dapu Li et al. “A efficient algorithm for molecular dynamics simulation on hybrid CPU-GPU computing platforms”. In: *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*. IEEE. 2016, pp. 1357–1363.

[78] Zhiyuan Li, Pen-Chung Yew, and Chuag-Qi Zhu. “Data dependence analysis on multi-dimensional array references”. In: *Proceedings of the 3rd international conference on Supercomputing*. ACM. 1989, pp. 215–224.

[79] Fabian Lorig et al. “Measuring and comparing scalability of agent-based simulation frameworks”. In: *German Conference on Multiagent System Technologies*. Springer. 2015, pp. 42–60.

[80] Sean Luke. “Multiagent simulation and the MASON library”. In: *George Mason University* 1 (2011).

[81] Sean Luke et al. “Mason: A multiagent simulation environment”. In: *Simulation* 81.7 (2005), pp. 517–527.

[82] Sean Luke et al. “Mason: A new multi-agent simulation toolkit”. In: *Proceedings of the 2004 Swarmfest Workshop*. Vol. 8. 2. Michigan, USA. 2004, pp. 316–327.

[83] Mikola Lysenko, Roshan M D’Souza, et al. “A framework for megascale agent based model simulations on graphics processing units”. In: *Journal of Artificial Societies and Social Simulation* 11.4 (2008), p. 10.

[84] Steven L Lytinen and Steven F Railsback. “The evolution of agent-based simulation platforms: a review of NetLogo 5.0 and ReLogo”. In: *Proceedings of the Fourth International Symposium on Agent-based Modeling and Simulation*. 2012, p. 19.

[85] C Macal, D Sallach, and M North. “Emergent structures from trust relationships in supply chains”. In: *Proc. Agent 2004: Conf. on Social Dynamics*. 2004, pp. 7–9.

[86] Charles M Macal and Michael J North. “Agent-based modeling and simulation: ABMS examples”. In: *Proceedings of the 40th Conference on Winter Simulation*. Winter Simulation Conference. 2008, pp. 101–112.

[87] Charles M Macal and Michael J North. “Tutorial on agent-based modelling and simulation”. In: *Journal of simulation* 4.3 (2010), pp. 151–162.

[88] Philip Machanick. *Approaches to addressing the memory wall*. Tech. rep. School of IT and Electrical Engineering, University of Queensland, 2002.

[89] Steven M Manson. “Agent-based modeling and genetic programming for modeling land change in the Southern Yucatan Peninsular Region of Mexico”. In: *Agriculture, Ecosystems & Environment* 111.1-4 (2005), pp. 47–62.

[90] Loris Marchal et al. “Parallel scheduling of DAGs under memory constraints”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 204–213.

[91] Alessio Marchetti. *Hyperlinked C++ BNF Grammar*. Oct. 29, 2018. URL: http://nongnu.org/hcb/#original-namespace-name (visited on 08/24/2019).

[92] Adam J McLane et al. “The role of agent-based models in wildlife ecology and management”. In: *Ecological Modelling* 222.8 (2011), pp. 1544–1556.

[93] Ruth Meyer. “Event-driven multi-agent simulation”. In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Springer. 2014, pp. 3–16.

[94] Fabien Michel. “Translating Agent Perception Computations into Environmental Processes in Multi-Agent-Based Simulations: A means for Integrating Graphics Processing Unit Programming within Usual Agent-Based Simulation Platforms”. In: *Systems Research and Behavioral Science* 30.6 (2013), pp. 703–715.

[95] 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.

[96] Nelson Minar et al. “The swarm simulation system: A toolkit for building multi-agent simulations”. In: (1996).

[97] Rob Minson and Georgios K Theodoropoulos. “Distributing RePast agent-based simulations with HLA”. In: *Concurrency and Computation: Practice and Experience* 20.10 (2008), pp. 1225–1256.

[98] Nitin Kumar Mishra and Nishchol Mishra. “Load Balancing Techniques: Need, Objectives and Major Challenges in Cloud Computing-A Systematic Review”. In: *International Journal of Computer Applications* 131.18 (2015).

[99] Daniel Moyo et al. “Macrophage Transactivation for chemokine Production identified as a negative regulator of granulomatous inflammation Using agent-Based Modeling”. In: *Frontiers in Immunology* 9 (2018), p. 637.

[100] Michael J North and Charles M Macal. *Managing business complexity: discovering strategic solutions with agent-based modeling and simulation*. Oxford University Press, 2007.

[101] Michael J North et al. *Visual agent-based model development with repast simphony*. Tech. rep. Tech. rep., Argonne National Laboratory, 2007.

[102] J. Oosten. *CUDA Thread Execution Model. 3D Game Engine Programming*. 2011. URL: https://www.3dgep.com/cuda-thread-execution-model/ (visited on 09/28/2018).

[103] John D Owens et al. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[104] Dawn C Parker and Vicky Meretsky. "Measuring pattern outcomes in an agent-based model of edge-effect externalities using spatial metrics". In: *Agriculture, Ecosystems & Environment* 101.2-3 (2004), pp. 233–250.

[105] 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*. Springer, 2012, pp. 271–308.

[106] H Van Dyke Parunak, Robert Savit, and Rick L Riolo. "Agent-based modeling vs. equation-based modeling: A case study and users' guide". In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Springer. 1998, pp. 10–25.

[107] Monali Patil and Vandana Jagtap. "Survey of Different Data Dependence Analysis Techniques". In: *International Journal of Advanced Engineering, Management and Science* 2.7 ().

[108] Roman Pavlov and Jörg P Müller. "Multi-agent systems meet GPU: deploying agent-based architectures on graphics processors". In: *Doctoral Conference on Computing, Electrical and Industrial Systems*. Springer. 2013, pp. 115–122.

[109] Josep M Perez, Rosa M Badia, and Jesus Labarta. "A dependency-aware task-based programming environment for multi-core architectures". In: *2008 IEEE International Conference on Cluster Computing*. IEEE. 2008, pp. 142–151.

[110] Manisa Pipattanasomporn, Hassan Feroze, and Saifur Rahman. "Multi-agent systems in a distributed smart grid: Design and implementation". In: *2009 IEEE/PES Power Systems Conference and Exposition*. IEEE. 2009, pp. 1–8.

[111] Aske Plaat, Henri Bal, and Rutger Hofman. *Bandwidth and Latency Sensitivity of Parallel Applications in a Wide-Area System*. 1998.

[112] Fiona Polack and Alastair Droop. "Principled simulation of cell proliferation dynamics using the CoSMoS approach". In: *Natural Computing* 14.1 (2015), pp. 63–82.

[113] Kleanthis Psarris and Konstantinos Kyriakopoulos. "Data dependence testing in practice". In: *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE. 1999, pp. 264–273.

[114] Kleanthis Psarris and Konstantinos Kyriakopoulos. "The impact of data dependence analysis on compilation and program parallelization". In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM. 2003, pp. 205–214.

[115] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE. 1991, pp. 4–13.

[116] Steve Railsback, Steve Lytinen, and Volker Grimm. "StupidModel and extensions: A template and teaching tool for agent-based modeling platforms". In: *Swarm Development Group. http://condor. depaul. edu/˜ slytinen/abm* (2005).

[117] Steven F Railsback, Steven L Lytinen, and Stephen K Jackson. "Agent-based simulation platforms: Review and development recommendations". In: *Simulation* 82.9 (2006), pp. 609–623.

[118] Mark N Read. "Statistical and modelling techniques to build confidence in the investigation of immunology through agent-based simulation". PhD thesis. University of York, 2011.

[119] Craig Reynolds. "Big fast crowds on ps3". In: *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. ACM. 2006, pp. 113–121.

[120] Craig Reynolds. “Boids background and update”. In: *http://www. red3d. com/cwr/boids/* (2001).

[121] Craig W Reynolds. “Flocks, herds and schools: A distributed behavioral model”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 21. 4. ACM. 1987, pp. 25–34.

[122] P Richmond. “Flame gpu technical report and user guide”. In: *Department of Computer Science Technical Report CS-11-03* (2011).

[123] Paul and Richmond. “Feasibility Study of Multi-Agent Simulation at the Cellular Level with FLAME GPU.” In: *FLAIRS Conference*. 2016, pp. 398–403.

[124] Paul Richmond and Mozhgan K Chimeh. “Flame GPU: Complex system simulation framework”. In: *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2017, pp. 11–17.

[125] Paul Richmond, Simon Coakley, and Daniela Romano. “Cellular level agent based modelling on the graphics processing unit”. In: *High Performance Computational Systems Biology, 2009. HIBI’09. International Workshop on*. IEEE. 2009, pp. 43–50.

[126] 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*. International Foundation for Autonomous Agents and Multiagent Systems. 2009, pp. 1125–1126.

[127] Paul Richmond and Daniela Romano. “Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu”. In: *Proceedings International Workshop on Supervisualisation*. 2008.

[128] Paul Richmond and Daniela Romano. “Template-driven agent-based modeling and simulation with CUDA”. In: *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 313–324.

[129] Paul Richmond et al. “High performance cellular level agent-based simulation with FLAME for the GPU”. In: *Briefings in Bioinformatics* 11.3 (2010), pp. 334–347.

[130] Duncan A Robertson. “Agent-based models to manage the complex”. In: *Managing Organizational Complexity: Philosophy, Theory, and Application* 24 (2005), pp. 417–430.

[131] Rebecca J Rockett et al. “Revealing COVID-19 transmission in Australia by SARS-CoV-2 genome sequencing and agent-based modeling”. In: *Nature Medicine* 26.9 (2020), pp. 1398–1404.

[132] Alban Rousset et al. “A survey on parallel and distributed multi-agent systems”. In: *European Conference on Parallel Processing*. Springer. 2014, pp. 371–382.

[133] Alban Rousset et al. “A survey on parallel and distributed multi-agent systems for high performance computing simulations”. In: *Computer Science Review* 22 (2016), pp. 27–46.

[134] Sergio Ruiz et al. “Reducing memory requirements for diverse animated crowds”. In: *Proceedings of Motion on Games*. 2013, pp. 77–86.

[135] Giulia Russo et al. “In Silico Trial to test COVID-19 candidate vaccines: a case study with UISS platform”. In: *BMC Bioinformatics* 21.17 (2020), pp. 1–16.

[136] Petrônio CL Silva et al. “COVID-ABS: An agent-based model of COVID-19 epidemic to simulate health and economic effects of social distancing interventions”. In: *Chaos, Solitons & Fractals* 139 (2020), p. 110088.

[137] Karandeep Singh and Chang-Won Ahn. “An agent based model approach for perusal of social dynamics”. In: *IEEE Access* 6 (2018), pp. 36948–36965.

[138] Prabhat Kr Srivastava, Sonu Gupta, and Dheerendra Singh Yadav. “Improving performance in load balancing problem on the grid computing system”. In: *International Journal of Computer Applications (0975–8887) Volume* (2011).

[139] Russell K Standish. “Going stupid with EcoLab”. In: *Simulation* 84.12 (2008), pp. 611–618.

[140] Ching-Lung Su et al. “Overview and comparison of OpenCL and CUDA technology for GPGPU”. In: *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*. IEEE. 2012, pp. 448–451.

[141] Sreenivas R Sukumar and James J Nutaro. "Agent-based vs. equation-based epidemiological models: A model selection case study". In: *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*. IEEE. 2012, pp. 74–79.

[142] Tao Sun et al. "An integrated systems biology approach to understanding the rules of keratinocyte colony formation". In: *Journal of the Royal Society Interface* 4.17 (2007), pp. 1077–1092.

[143] Vinoth Suryanarayanan and Georgios Theodoropoulos. "Synchronised range queries in distributed simulations of multiagent systems". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 23.4 (2013), p. 25.

[144] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. "Pdes-mas: Distributed simulation of multi-agent systems". In: *Procedia Computer Science* 18 (2013), pp. 671–681.

[145] Shailesh Tamrakar. "Performance optimization and statistical analysis of basic immune simulator (BIS) using the FLAME GPU environment". In: (2015).

[146] Masahiro Tanaka and Osamu Tatebe. "Workflow scheduling to minimize data movement using multi-constraint graph partitioning". In: *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society. 2012, pp. 65–72.

[147] Yuqing Tang, Simon Parsons, and Elizabeth Sklar. "Modeling human education data: From equation-based modeling to agent-based modeling". In: *Multi-Agent-Based Simulation VII*. Springer, 2007, pp. 41–56.

[148] Eric Tatara et al. "An indroduction to repast simphony modeling using a simple predator-prey example". In: *Proceedings of the Agent 2006 Conference on Social Agents: Results and Prospects*. 2006.

[149] Wanwimol Thawornchak. "Equation-Based and Agent-Based Modeling of Supply Networks". In: (2001).

[150] Georgios Theodoropoulos and Brian Logan. "The Distributed Simulation of Agent-Based Systems". In: *IEEE Proceedings Journal, Special*

Issue on Agent-Oriented Software Approaches in Distributed Modeling and Simulation. Citeseer. 2001.

[151] Georgios Theodoropoulos et al. "Large scale distributed simulation on the grid". In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Vol. 2. IEEE. 2006, pp. 63–63.

[152] Seth Tisue and Uri Wilensky. "NetLogo: Design and implementation of a multi-agent modeling environment". In: *Proceedings of Agent*. Vol. 2004. 2004, pp. 7–9.

[153] Banerjee Utpal K. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[154] Banerjee Utpal K. "Speedup of Ordinary Programs." PhD thesis. University of Illinois at Urbana-Champaign, 1979.

[155] WF Van Gunsteren and HJC Berendsen. "Algorithms for Brownian dynamics". In: *Molecular Physics* 45.3 (1982), pp. 637–647.

[156] DC Walker et al. "The epitheliome: agent-based modelling of the social behaviour of cells". In: *Biosystems* 76.1-3 (2004), pp. 89–100.

[157] Mitchell Welch et al. "Improving the efficiency of large-scale agent-based models using compression techniques". In: *Multidisciplinary Computational Intelligence Techniques: Applications in Business, Engineering, and Medicine*. IGI Global, 2012, pp. 301–326.

[158] Richard A Williams. "User experiences using FLAME: A Case study modelling conflict in large enterprise system implementations". In: *Simulation Modelling Practice and Theory* 106 (2020), p. 102196.

[159] Michael Wolfe and C-W Tseng. "The power test for data dependence". In: *IEEE Transactions on Parallel and Distributed Systems* 3.5 (1992), pp. 591–601.

[160] Jia-Hwa Wu and Chih-Ping Chu. "The quadratic test: And exact data dependence test for quadratic expressions". In: *Compiler Techniques for High-Performance Computing (CTHPC'03)* (2003).

[161] Fabian Ying and Neave O'Clery. "Modelling COVID-19 transmission in supermarkets using an agent-based model". In: *arXiv preprint arXiv:2010.07868* (2020).

[162] Hong Zheng et al. *A primer for agent-based simulation and modeling in transportation applications*. Tech. rep. United States. Federal Highway Administration, 2013.

[163] Anja Zöller et al. "Benchmarking of Multiagent Systems". In: *Multiagent Engineering*. Springer, 2006, pp. 557–574.