Supporting agent-based simulations on GPU

Li, Xiaosong

2016

Li, X. (2016). Supporting agent-based simulations on GPU. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/69413>

<https://doi.org/10.32657/10356/69413>

NANYANG TECHNOLOGICAL UNIVERSITY

Supporting Agent-based Simulations on GPU

By

Li Xiaosong

Supervisor: Professor Cai Wentong

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2016

Abstract

Using agent-based simulation (ABS) to analyze complex systems gains growing popularity over the past decades. It allows to take into account different levels of interactions as well as the heterogeneity of agents in the system. Through ABS, it is possible to forecast and explore future scenarios, experiment possible alternative decisions, and set different values for the decision variables to analyze the effects of these changes. At an aggregated level, the use of ABS can help in understanding general properties and patterns concerning the whole scenario that could not be deduced nor forecasted by the observation of each individual agent, due to the complexity of the interactions occurring among the agents.

One of the fundamental issues in ABS is the speed of execution. As ABS is gaining a more widespread adoption, the need to model agents with more complex and advanced behavior is on the rise. Besides, certain phenomena can only be observed in scenarios with large number of agents, and the number of simulation instances can also be large when various possibilities of the scenario are explored. The complexity and large-scale of ABS pose challenges to the execution efficiency of ABS.

In this research, effective and generic strategies are proposed to speed up the ABS execution with GPU platform. The strategies are proposed for the execution of a single ABS execution as well as a parameter space exploration task consisting of multiple ABS instances. The GPU is leveraged in the proposed strategies to exploit the parallelism of ABS.

First, two common modules that widely exist in ABS applications are identified, namely the agent management module and the agent interaction module. Improving the efficiency of these two common modules can speed up the ABS execution in general. To support the two modules on GPU, strategies that use an AgentPool data structure to handle agent creation and deletion and GPU memory hierarchy to support efficient agent interaction are proposed. Experiment results show the performance advantage of the proposed strategies over FLAME, a well established GPU based ABS framework.

Second, a GPU based ABS library consisting of a set of easy-to-use APIs is provided to facilitate the implementation of ABS using the proposed modules. To demonstrate the effectiveness and generality, the library is applied to implement a range of applications, including game-of-life, flocking boids, prey-and-predator, and the social-force based crowd simulation. The simulation results demonstrate that the proposed modules can be applied to various scenarios and achieve better performance than the commonly used CPU and GPU ABS frameworks, namely MASON and FLAME, for ABS applications using continuous space.

Third, a simulation cloning technique for ABS, namely the top-down cloning strategy, is proposed to accelerate a parameter space exploration task consisting of multiple simulation instances. The top-down cloning strategy reuses common computations among simulation instances while maintaining the correctness of execution. Its GPU implementation supports the concurrent execution of agents within each simulation instance as well as the concurrent execution of multiple simulation instances. The performance of the proposed approach is evaluated using a case study of an evacuation scenario. The experiment results demonstrate that the top-down cloning strategy can shorten the overall execution time of the parameter space exploration task.

Fourth, the idea of the ABS cloning is further developed by removing the limitations of the top-down cloning strategy. To overcome the limitations, an improved strategy named the bottom-up cloning strategy is proposed. In the experiments, a parameter exploration task consisting of a large number of simulation runs is executed with the bottom-up cloning strategy. It shows that the bottom-up cloning strategy can effectively shorten the execution time of the parameter space exploration and support more scenarios than the top-down cloning strategy.

In summary, the contributions of this research are listed as follows: (1) The proposed GPU based agent management module and agent interaction module achieve better execution efficiency than the commonly used GPU based ABS framework for the execution of a single ABS instance; (2) The proposed GPU based ABS library can be generically applied to various ABS applications and accelerate their executions; and (3) The top-down and the bottom-up cloning strategies can speed up a parameter space exploration task that consists of multiple ABS instances. The bottom-up cloning strategy can support broader range of applications than the top-down cloning strategy.

Acknowledgments

First and foremost, I would like to express my earnest and deepest gratitude to my PhD supervisor, Professor Cai Wentong, for his remarkable guidance, constant help and support, and lots of caring and patience in the past five years. He is a mentor of a lifetime to me, and I am honored to be his student. I believe that his immense knowledge, passion, and rigorous attitude in research not only help me completing this thesis, but will also influence my career and life in the future.

I would like to thank my parents, grandparents, Aunt Qing, Cousin Ruixu, and other family members for their love, encouragements and confidences in me. Especially, I would like to thank my girlfriend, Jingyi. We have been in a long distance relationship for over three years, and she has always been patient and supportive, and looking forward for the reunion. Taking a PhD is not easy. They give me the strength to carry on.

I would like to acknowledge the help and support provided by the wonderful professors and staffs in School of Computer Science and Engineering. I appreciate Professor Stephen John Turner for his guidance as my co-supervisor in the first three and half years of my PhD study. I appreciate Professor Tang Xueyan for offering me a research assistant position to support my last year of the study. I have been also learning a lot from him in our weekly meetings. I appreciate Ms. Ng-Goh Siew Lai Irene for her great skills of building the experiment test bed. I also want to express my thanks to Professor Sun Chengzhen, Professor He Bingsheng, Professor Xing Zhenchang, and many other professors for their help in many aspects.

Special thanks go to my friends and colleagues in Singapore: Dr. Li Yusen, Dr. Xu Yi, Dr. Zhong Jinghui, Dr. Zhou Chi, Dr. Tang Shanjiang, Dr. Wang Zeke, Dr. Ou Jiajue, Dr. Deng Yunhua, Dr. Li Zengxiang, Dr. Liu Cheng, Mr. Tan Wen Jun, Ms. Zhao Mingbi, Mr. Xu Yadong, Ms. Yin Haiyan, Mr. Guan Yitong, Mr. Ren Runtian,

Mr. Liu Yi, Dr. Zhang Wei, Dr. Liu Fang, Dr. Lai Pan, and many others, for their companionship. Cheers to the joyfulness and sorrowfulness that we shared together.

Last but not least, I am grateful to Nanyang Technological University and School of Computer Science and Engineering for providing me the opportunity and scholarship to explore research and complete the PhD program.

Contents

Abstract	i
Acknowledgments	iii
List of Tables	ix
List of Figures	x
Glossary	xiv
Acronyms	xv
1 Introduction	1
1.1 Background	1
1.2 Motivation	6
1.2.1 Prevailing trend of using ABS to analyze complex systems	6
1.2.2 Inefficiency of ABS Execution	7
1.2.3 Opportunities and challenges of adopting GPU in ABS	8
1.3 Objectives	10
1.3.1 Accelerate a single ABS instance	11
1.3.2 Accelerate multiple ABS instances in a parameter space exploration task	11
1.4 Contributions	12
1.5 Structure of the thesis	13
2 Literature Survey	15
2.1 Using GPU as an accelerator in general	15
2.2 Execution strategy for a single ABS instance	17
2.2.1 Typical ABS toolkits	18
2.2.2 Strategies to accelerate ABS with GPU	19

2.3	Execution strategy for a parameter space exploration task	23
2.3.1	Parameter space exploration	23
2.3.2	Accelerating parameter space exploration task	24
2.3.3	Simulation cloning techniques	25
2.4	Summary	27
3	Supporting Common Modules of ABS on GPU	29
3.1	Overview	29
3.2	Challenges of executing ABS on GPU	30
3.3	Strategy implementation	31
3.3.1	Agent management	32
3.3.2	Agent interaction	35
3.4	Experiments	43
3.4.1	Agent management	44
3.4.2	Agent interaction	45
3.4.3	Comparison with FLAME	48
3.5	Summary	48
4	A Library Supporting ABS on GPU	51
4.1	Overview	51
4.2	GSim library	52
4.2.1	Classes	52
4.2.2	Workflow	56
4.3	Analysis of usability	58
4.3.1	FLAME workflow	58
4.3.2	Usability comparison	59
4.4	Experiments	62
4.5	Summary	66

5	A Top-down ABS Cloning Strategy	69
5.1	Overview	69
5.2	Cloning agent-based simulations	71
5.2.1	Characteristics of cloning agent-based simulations	71
5.2.2	Incremental cloning of agent-based simulations	72
5.2.3	Top-down cloning strategy	74
5.3	GPU implementation	78
5.3.1	Overall workflow	78
5.3.2	Cloning condition checking	79
5.3.3	Actual agent creation	81
5.3.4	Throttling mechanism	82
5.3.5	Optimization	83
5.4	Experiments	84
5.4.1	Verification	85
5.4.2	Benefit of optimization	86
5.4.3	Overall performance gain	88
5.5	Summary	89
6	A Bottom-up ABS Cloning Strategy	91
6.1	Overview	91
6.2	Issues of the top-down cloning strategy	92
6.3	Bottom-up cloning strategy	94
6.3.1	Constructing cloning tree - an MST heuristic	95
6.3.2	Active cloning and passive cloning	96
6.4	GPU implementation	97
6.4.1	Overall workflow	98
6.4.2	Cloning decision making and actual agent creation	99
6.5	Experiments	101
6.5.1	Verification	102
6.5.2	Cloning vs. sequential stand-alone execution	104
6.5.3	Throttling mechanism	107

6.5.4	Bottom-up cloning tree evaluation	107
6.5.5	Cloning vs. parallel stand-alone execution	108
6.5.6	Large scale case studies	110
6.6	Summary	111
7	Conclusions and Future Work	113
7.1	Conclusions	113
7.2	Future Work	115
7.2.1	Enhancing the single ABS execution on GPU	115
7.2.2	Enhancing the ABS cloning	116
	Author's Publications	119
	References	121

List of Tables

3.1	<code>AgentPool</code> arrays	32
3.2	<code>Context</code> arrays	36
3.3	Experiment setups	43
3.4	Profiling neighbor searching strategies	46
4.1	Module descriptions	56
4.2	Experiment setups	63
4.3	Characteristics of general ABS [84]	64
5.1	<code>AgentPool</code> and <code>Context</code> of scenario in Figure 5.1	77
5.2	Experiment setups	85
5.3	Profiled execution time of various optimization cases at a time step	86
6.1	Experiment setups	102

List of Figures

1.1	A typical agent structure [65]	2
1.2	GPU architecture [74]	5
1.3	Four pieces of work in this research and their relationship	12
3.1	Demonstration of AgentPool manipulation	33
3.2	The basic neighbor searching strategy on GPU	37
3.3	The naive neighbor sharing strategy	39
3.4	The enhanced neighbor sharing strategy	40
3.5	Performance comparison on memory management	45
3.6	Performance comparison among PGM, NNS and ENS	47
3.7	Comparison between ENS and FLAME in agent interaction	48
4.1	GSim classes	53
4.2	Example ConstantData and RuntimeData	54
4.3	AgentPool.stepPool function	55
4.4	The AgentPool.swap function	55
4.5	Using Context 's neighbor searching APIs to realize agent interaction	56
4.6	A flocking boid application implemented with GSim	57
4.7	A flocking boid application implemented with FLAME, XML file	60
4.8	A flocking boid application implemented with FLAME, agent function	61
4.9	A flocking boid application implemented with FLAME, the init file	62
4.10	Performance comparison on various applications	65
5.1	Cloning tree and computation sharing	73
5.2	Groups of concurrent clones and synchronization barriers	83

5.3	Scenario of the case study	85
5.4	Correctness verification	85
5.5	Performance of optimization	87
5.6	Execution time and speedup vs. number of agents	89
5.7	Execution time and speedup vs. number of clones	90
6.1	A scenario that top-down cloning strategy is not effective	93
6.2	The same scenario of Figure 6.1 handled by the bottom-up cloning strategy	96
6.3	Scenarios and their corresponding cloning trees	103
6.4	Correctness verification: Scenario 1	104
6.5	Correctness verification: Scenario 2	105
6.6	Comparing cloning strategies with stand-alone execution	106
6.7	Throttling mechanism	107
6.8	Cloning tree evaluation	108
6.9	Bottom-up cloning strategy vs. parallel stand-alone execution on GPU .	109
6.10	Example parameter setups of the large scenario	110
6.11	Performance of the bottom-up cloning strategy on three platforms	111

Glossary

AI_i is an index for agent a_i to access its data in the auxiliary arrays of the AgentPool.

$C(v_1, v_2, \dots, v_{n_p})$ represents the simulation instance that evaluate parameter value combination $(v_1, v_2, \dots, v_{n_p})$.

CI_i represents cell id.

DI_i is an index for agent a_i to access its data in the data arrays of the AgentPool.

EI_i is the index of the reference to the last agent of i^{th} cell in the agentPtrArray.

N_{all} is the total number of agents in the simulation.

N_{block} is the number of threads per block.

N_{smem} is the number of agents that the shared memory can accommodate.

$Rect_{block}(u_b, l_b, d_b, r_b)$ is the bounding box that encompasses $Rect_i(u_i, l_i, d_i, r_i)$ of the agents that belong to the same block.

$Rect_i(u_i, l_i, d_i, r_i)$ is the minimal rectangle formed by cells that overlap with the agent's interaction range.

SI_i is the index of the reference to the first agent of i^{th} cell in the agentPtrArray.

$a_j^{(v_1, v_2, \dots, v_{n_p})}$ represents the cloned version of the j^{th} agent in simulation instance $C(v_1, v_2, \dots, v_{n_p})$.

n_i is the number of value choices of the i^{th} parameter.

n_p is the total number of parameters in a parameter space exploration task.

p_i is the i^{th} parameter.

v_i is the value choice of the i^{th} parameter.

Active cloning takes place when an agent senses parameter in its sensing range.

AgentPool stores one type of agents.

Cloned simulation instance is a simulation instance that is cloned from another simulation instance.

Cloning tree is a tree structure recording the parent-child clone relationship. Child clone can reuse the computation of parent clone.

Context stores references to agents that can be accessed in one simulation instance.

Incremental cloning means agents are cloned incrementally when necessary.

Passive cloning takes place when an agent that is not yet cloned interacts with cloned agents.

Simulation scenario is defined by parameter value combination (or configuration) of a simulation model.

Stand-alone simulation instance is a simulation instance that contains all agents. It executes independently and does not share computation with any other simulation instance.

Acronyms

ABS Agent-based Simulation

CA Cellular Automata

CUDA Compute Unified Device Architecture

ENS Enhanced Neighbor Sharing

FLAME Flexible Large Scale Modelling Environment

GPU Graphics Processing Unit

HLA High-level Architecture

MASON Multi-Agent Simulator of Neighborhoods

MST Minimum Spanning Tree

NNS Naive Neighbor Sharing

PDES Parallel Discrete Event Simulation

PGM Pure Global Memory

SIMT Single-Instruction-Multiple-Threads

SM Stream Multi-processor

SP Stream Processor

Chapter 1

Introduction

This chapter provides the background knowledge of agent-based simulation (ABS), parameter space exploration, and Graphics Processing Unit (GPU) architecture. The motivations of this research are presented, namely the prevailing trend of using ABS to model complex systems, the execution inefficiency of ABS due to its large scale and complex agent model, and the opportunities and challenges to accelerate ABS on GPU platform. The research aims, contributions, and thesis structure are also summarized.

1.1 Background

Supporting and accelerating agent-based simulation on GPU requires the understanding of both the characteristics of agent-based simulation and the features of GPU architecture. In this section, the important concepts in this thesis are explained, namely the agent-based simulation, parameter space exploration, and GPU architecture.

Agent-based simulation: Complex systems usually have many interacting entities and non-linear interactions among them. The ABS is a powerful tool to analyze complex systems, as it can generate very complex collective behaviors by using a number of simple interacting rules between communicating autonomous entities, i.e., agents. A synonym of ABS would be microscopic modeling. Opposite to microscopic modeling would be macroscopic modeling [9], or aggregate modeling. One of the major limitation of macroscopic models is that complex emergent phenomena cannot be captured easily [55]. Besides, the macroscopic models do not explicitly consider individual variations in the behavioral rules (heterogeneity) and random influences of variations (stochasticity)

Figure 1.1: A typical agent structure [65]

[40], and they have also been criticized for not being illustrative and explanatory to describe emergent phenomena. In contrast, in the ABS which is a bottom-up approach, the behavior of individual agent can be generally and intuitively specified through rules, such as if-then kind of rules or logical operations. The heterogeneity and stochasticity can be also taken into consideration. The combined interactions of all agents can create rich emergent behaviors. Patterns, structures, and behaviors of such systems are not explicitly specified in the agent-based models, but arise through agent interactions.

A typical agent-based model has three elements [65] as illustrated in Figure 1.1:

- (i) A set of agents with their own properties and behaviors;
- (ii) The relationship determining the interactions among agents; and
- (iii) The environment where the agents situate.

Regarding the agent, the most important defining characteristic of an agent is its capability to act *autonomously*, that is, to act on its own without external direction in response to situations it encounters. Agents maintain their own properties and are endowed with behaviors that allow them to make independent decisions. The execution mechanism of agents is usually based on time-step (or frame). At each time-step, all agents concurrently and autonomously perform their behaviors and update their states.

Regarding the agent interaction, agents are *interactive* and have protocols for interaction with other agents, such as for communication, movement and collision avoidance. An agent's neighborhood is a general concept applicable to whatever agent spaces that are defined in the model. For example, an agent can interact only with its neighbors located close-by in physical or geographical space, or an agent can interact with neighboring agents connected in a social network.

Regarding the agent environment, agents also interact with the *environment*. The environment may simply be used to provide information on the spatial location of an agent relative to other agents, or it may affect agent behaviors, such as obstacles that affect an agent movement. The environment can be continuous space, discrete cell-based space, or based on the network. Different types of environment also affect how agents interact. The interaction frequency and the influence strength of agents may depend on physical or geographical distance between agents. The interactions could also be based on logical connections between agents, which are usually modeled as social networks or other interaction networks.

Parameter space exploration: An agent-based model may consist multiple parameters. The parameters can be agent properties (e.g., max velocity and sensing radius) or environment properties (e.g., placement and size of obstacles). The number of model parameters can be determined based on the purpose of the model and its required degree of realism and accuracy. A model of practical applications tends to have more parameters than a model aimed at the fundamental understanding of social or economic mechanisms.

To study the effects of these parameters, parameter space exploration task is usually an iterative process and often involves a large number of simulation runs. In each simulation instance, one parameter configuration is tested and a fitness value of this parameter configuration is generated. Fitness values of different parameter configurations are compared and synthesized to generate new combinations of the parameter values for the next round of processing, until the desired parameter configuration is found. Simply put, parameter space exploration tries some values and simulates the model to see how the model behaves globally [10]. The exploration of the desired parameter setting of an agent-based model can become long and tedious if the parameter space is not explored in an efficient and a systematic manner.

In general, the parameter space exploration is to filter out the non-optimal parameter configurations until the optimal solution is found. The term “optimal” is defined case by case. The workflow of a parameter space exploration can be summarized as the pipeline of configuration generator, execution engine, and optimal solution selector. In detail, the generator generates various parameter configurations according to the inputs. The engine produces simulation output with each combination as an input. The selector analyzes the produced results, picks out those configurations that generate results closer to the optimization goal, and feeds them as the inputs of the next round of configuration generator, hopefully to produce better results.

The purposes of the evaluation may vary. For example, parameter space exploration could be used to evaluate different design alternatives (and to answer “what-if” questions), analyze sensitivity of simulation parameters [30], calibrate model parameters [10, 95], or identify simulation models which exhibit emergent system behaviors of interest [103, 113]. The parameter exploration is widely adopted in economics, logistics, military, sociology, and engineering design. For example, parameter space exploration is adopted in a simulation-based military methodology utilized to uncover weakness of tactical plans, known as the Automated Red Teaming (ART) [26].

GPU architecture: Our execution strategies proposed for ABS and parameter space exploration task are designed for and conducted on GPU platform. GPU is capable of carrying out a huge number of simple workloads. Taking NVIDIA’s GPU as an example, NVIDIA provides a parallel computing platform and application programming interface model, known as Compute Unified Device Architecture (CUDA). On the programming side, the CUDA programming language resembles C/C++ programming. On the hardware side, in CUDA enabled GPU, the lowest level processing unit is the stream processor (SP) with fully pipelined floating-point and integer arithmetic logic units (ALUs). Multiple SPs form a stream multi-processor (SM). An SM is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs an execution model called Single-Instruction-Multiple-Threads (SIMT). The multiprocessor creates, manages, schedules and executes in groups of 32 parallel threads called warps. Multiple warps form a block and multiple blocks form a kernel grid. When a CUDA program on the host (CPU side) invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. Multiple

Figure 1.2: GPU architecture [74]

blocks can be assigned to one multiprocessor and these blocks are executed in parallel. As blocks terminate, new blocks are launched on the vacated multiprocessors [74].

Figure 1.2(a) shows the architecture of an NVIDIA CUDA-enabled GPU in comparison with CPU. Physically, a GPU consists of dynamic random-access memory (DRAM) and many SMs. Each SM has multiple SPs. Each SM can support up to 2048 concurrent threads. The difference between CPU and GPU is that GPU has much more SPs (ALUs) than CPU.

The NVIDIA CUDA GPU has a hierarchical memory design as shown in Figure 1.2(b). The global memory, constant memory, and texture memory are accessible by all threads. The global memory can be read and written but the access speed is slow. Compared with global memory, the constant memory, and texture memory are read-only but have higher bandwidth because of their caching policy. Besides, the size of the constant memory is limited, and the texture memory is designed for graphics applications rather than general purpose applications. The per-block shared memory can only be accessed within a thread block and is of limited size, up to 48 kilobytes per block. It is small but has about seven times higher bandwidth than that of the global memory under full utilization [100].

1.2 Motivation

The motivation of carrying out this research is that using ABS is prevailing in analyzing complex systems. However, the execution of ABS can be time consuming because of its large scale and complex agent interaction. The total execution time grows even more if the number of simulation instances is large in a parameter space exploration task consisting of multiple simulation instances to be executed. Obtaining the simulation results faster means more time for the decision making. GPU platform is a commodity accelerator for various applications¹. In order to facilitate the system analysis and decision making, the opportunities and challenges of using GPU platform to accelerate ABS are therefore studied.

1.2.1 Prevailing trend of using ABS to analyze complex systems

With years of development, ABS has become a mainstream method of modeling and simulation [71]. ABS is very versatile and its usage is prevailing. It can be applied to mainly four categories of problems [9]: flows, markets, organizations, and diffusion. The category of flows includes evacuation [41], traffic [7], and customer flow management [99]. The category of markets includes stock market [5], customer behavior [72], and strategy simulation [66]. The category of organization includes the applications of operational risk [20, 44] and operational design analysis [56]. The diffusion category includes epidemics [28], and the diffusion of innovation adoption dynamics [6, 11, 24].

The benefits of ABS over other modeling techniques can be captured in three statements [9]: (i) ABS captures emergent phenomena; (ii) ABS provides a natural description of a system; and (iii) ABS is flexible.

ABS captures emergent phenomena: Emergent phenomena are the results of interactions of systems' parts and cannot be extracted individually. This characteristic of emergent phenomena makes them difficult to understand and predict: emergent phenomena can be counterintuitive. ABS is the canonical approach to modeling emergent phenomena. Emergence is captured from the bottom up when the simulation is run. For example, one branch of study in economics is called agent-based computational economics

¹<http://www.nvidia.com/object/gpu-applications.html>

(ACE) [97]. One area where ACE methodology has frequently been applied is asset pricing. Through ACE, researcher finds that the phenomena of large booms and busts in asset prices occur as agents switch across different asset price forecasting strategies [45].

ABS provides a natural description of a system: In many cases, ABS is most natural for describing and simulating a system composed of behavioral entities. It is more natural to describe rules and behaviors of agents than to come up with the equations that govern the dynamics of the whole system. For example, in the research of human crowd, the crowd can either be modeled as flow and studied with theories in fluid dynamics, or modeled with agent-based approach [115]. The flow based model basically neglects the features of individuals, including physical features (e.g., position and speed), social features (e.g., social norms, family ties and leadership) and psychological features (e.g., emotion). In contrast, ABS allows these individual features to be considered, therefore it generates more realistic results.

ABS is flexible: The flexibility of ABS can be observed along multiple dimensions. In one dimension, ABS allows the modeling of complex agent behavior, (e.g., the ability to learn and evolve, and the rules for interactions). Another dimension is the ability to change levels of description and aggregation. ABS can consist of single agents, groups of agents, and aggregated agents, and can have different levels of details coexisting. For example in [63], the agents are modeled with human-like decision making and behavior execution capabilities in crowd simulation. Particularly, the effect of group movement is studied (e.g., close friends or family members can form a group).

1.2.2 Inefficiency of ABS Execution

The complex adaptive system community tends to build large models to fully represent real systems. In order to capture the emergent phenomena, the scale of the simulation must reach certain level. Not only is the number of agents in a simulation large, but also the number of simulations performed in one study is also large.

Taking the crowd simulation for example, agent-based approach allows agents to have cognitive and reasoning capabilities when making decisions, therefore it incurs significantly more computation cost than flow-based approach. In fact, existing work on modeling huge-sized crowd (thousands of agents) usually treats the crowd as a whole

using flow-based approach and focuses on the global trend of the crowd, due to the tremendous computation cost involved [115]. For example, in an ABS study of evacuation from Beijing national stadium [18], the execution time of sequential execution is increased by 4.11 times (1976 seconds to 10097 seconds) when the number of agents increases from 3000 to 8000. When the number of agents increases beyond 8000, the simulation takes too long to finish. This shows that the execution of large scale ABS is computation demanding and inefficient.

The future study of complex adaptive system tends to be at the scale of “world model” level that the activities of agents worldwide will be taken into account [60]. The world model has even larger scale. Hence, making the execution of these models possible requires not only sufficient computing power but also efficient execution strategies.

Not only the number of agents could be large, the number of simulation instances could also be enormous. A parameter space exploration just with a few parameters may require a large number of simulations. Running a large number of simulations requires a significant amount of execution time. For example, parameter space exploration is adopted to uncover weakness of military operation plans in [26]. A tactic plan contains a number of parameters. Given the number of adjustable parameters and associated ranges, the search space contains 6×10^{23} parameter value combinations. The goal is to find a parameter value combination (a tactic plan) that maximize the enemy casualties. To achieve this goal, each parameter value combination (i.e., a plan) is evaluated using a simulation. To address the issue of huge number of simulations, a complex adaptive system evolver (CASE) framework is proposed to reduce the total number of simulation executions required from 6×10^{23} to 1500. Although the number of simulations is dramatically reduced, the whole exploration still takes 27 hours and 22 minutes to finish.

In general, obtaining the simulation results faster means more time for decision making. Due to the large number of agents and simulation runs, the execution of ABS is potentially time consuming. Therefore, strategies need to be proposed to increase the execution efficiency of ABS.

1.2.3 Opportunities and challenges of adopting GPU in ABS

Compared with macroscopic modeling (e.g., modeling crowd as a flow), ABS simulates the behaviors of individuals and is considerably more computationally demanding. How-

ever, the agent’s autonomy and individual processing also means ABS has the intrinsic parallelism, i.e., agents could be processed in parallel. Despite the obvious parallelism, majority of ABS applications fail to exploit it and are often implemented based on highly serialized algorithms for agent processing. Sequential execution places limitations on both the scale of models and the speed at which they may be simulated. Parallel and distributed platforms are commonly used to improve the execution performance of simulation. But, because of the huge investment for large scale clusters and the limitation of bandwidth between nodes in clusters, blindly increasing the cluster size and the number of CPU cores may not obtain the matching performance improvement to the investment.

GPU provides the opportunities to support and accelerate ABS, because of its advantages of being commodity hardware and having strong ability of parallel computing. Particularly, the parallel computing capability provided by GPU is suitable to accommodate the intrinsic parallelism of ABS. Mapping ABS applications that are running on serialized platforms onto parallel platform (e.g., GPU) and understanding the impact of architectural difference on algorithm design are increasingly important, but not yet well studied. The parallel implementation of the agent-based models on GPU has become a recent trend with significant performance improvement reported, compared to the serial counterpart on the CPUs, especially for extra-large scale simulations. The existing works will be reviewed in Chapter 2.

However, GPU programming requires a particular programming style because it relies on a highly specialized architecture. Indeed, because they are designed for graphics, GPUs are very restrictive in operations and programming and the hardware can only be used in certain ways. Several issues of GPU need to be taken into consideration in designing execution strategies: i) Compared with multi-core CPU, GPU provides much more parallel threads, but the clock rate of each thread is much slower; ii) Execution on GPU is only efficient if a single instruction issued on GPU can be executed by multiple threads simultaneously (SIMT programming paradigm); iii) GPU memory access pattern has determinant effect on the overall performance; and iv) GPU does not provide an efficient support for dynamic memory allocation. So, performance degrading may happen due to the lower clock speed of GPU per core and inefficient memory access pattern [104]. Hence, designing execution strategies for ABS on GPU is necessary, because GPU is only

useful and effective for problems which can be modeled with respect to its programming and architectural contexts.

Besides supporting the execution of a single ABS instance, strategies to execute multiple simulation instances in a parameter space exploration task is also investigated. Executing an ABS can take considerable amount of time. Exploring parameter space with agent-based model will take even longer time because the number of simulation instances can be also large.

The computation sharing among multiple simulation instances of the same model with different parameter configurations sheds light on speeding up the parameter space exploration task. The computation sharing in this thesis means that different simulation instances of the same model may share some common execution paths. So, computation can be saved by processing the common execution paths shared by multiple simulation instances only once. Therefore, the overall computation can be reduced and the total execution time can be shortened.

The challenges of proposing an effective strategy to exploit the computation sharing is that both the efficiency and correctness need to be guaranteed at the same time. The efficiency means that the strategy can actually reduce the simulation time, i.e., the extra work incurred by the strategy should be less than the saved computations. The correctness means that the simulation results obtained with or without the strategies should be the same. Besides, particularly on GPU platform, the execution of a parameter space exploration task has two-level parallelism, i.e., parallelism within a simulation instance and parallelism among multiple simulation instances. This two-level parallelism also needs to be taken into consideration when the strategies are designed.

1.3 Objectives

The major goal of our research is to design mechanisms to accelerate the execution of agent-based simulations. Strategies to accelerate both the executions of a single simulation instance and an agent-based parameter space exploration task consisting of multiple simulation instances of the same model will be proposed.

1.3.1 Accelerate a single ABS instance

For single instance, there are works aiming at accelerating ABS with GPU platform [27, 64, 86]. The cellular automata (CA) technique is usually chosen to model the environment and agent interactions on GPU because of the straightforward mapping of cells to GPU threads. ABS modeled in continuous space is another major type of ABS applications. The existing mechanisms to accelerate ABS modeled in CA with GPU platform are not directly applicable to ABS modeled in continuous space (to be explained in Section 2.2.2). Therefore, new acceleration mechanisms need to be proposed for this type of ABS.

The proposed strategy should be generic enough to be applied to a group of typical ABS applications. In the literature, there are works aiming at accelerating particular applications. Some of them are discussed in Chapter 2. In this research, our objective is to extract the common modules of a group of typical agent-based simulations and propose strategies to accelerate these modules so that ABS adopted these common modules can benefit. Another objective is to propose strategies that can be applied with minimal effort to the existing ABS applications for execution acceleration. To achieve this goal, common modules will be identified and optimized with our strategies while application dependent modules remain unchanged and can still be reused.

1.3.2 Accelerate multiple ABS instances in a parameter space exploration task

For parameter space exploration, strategies to exploit the computation sharing in multiple parallel discrete event simulation (PDES) instances have been studied in [50]. However, exploiting the computation sharing among multiple ABS instances has not been studied in the literature to the best of our knowledge. The computation sharing techniques for PDES [50] and distributed simulation based on high-level architecture (HLA) [14] cannot be directly applied to ABS (to be explained in Section 2.3.3). Therefore, new strategies need to be developed to effectively exploit and reuse the computation sharing of multiple ABS instances.

Figure 1.3: Four pieces of work in this research and their relationship

1.4 Contributions

In this thesis, four pieces of work are accomplished to achieve the effectiveness and generality goals. Figure 1.3 shows the overall structure of this thesis. The contributions of this thesis are summarized as follows:

Supporting common modules of ABS on GPU: By studying various ABS applications and toolkits, two modules that commonly exist in ABS applications are identified, namely the agent management module and the agent interaction module. Efficient strategies are proposed to support these two modules on GPU. The major issues of these two modules are the inefficiency of dynamic memory allocation and the inefficiency of random memory access. Our main contributions are: 1) an agent management module with an `AgentPool` data structure to support multiple types of agents and dynamic agent creation and deletion during the simulation on GPU; and 2) an agent interaction module which takes advantage of GPU memory hierarchy to accelerate the neighbor searching. Experiments are conducted to demonstrate the efficiency and effectiveness of our design of agent management module and agent interaction module.

A library supporting ABS on GPU: Migrating the existing agent models developed for CPU platform to GPU platform is not a trivial task and requires advanced GPU programming skills. To ease the difficulty of developing ABS on GPU platform, a library called GSim is developed to support the execution of generic ABS modules on GPU. APIs of agent management module and agent interaction module are provided. Four applications are implemented as case studies to compare the usability and efficiency of the proposed GSim library with the existing well developed toolkits on both CPU and GPU

platform, namely Multi-Agent Simulator of Neighborhoods (MASON) [61] and Flexible Large Scale Modelling Environment (FLAME) [87] (on NVIDIA spotlight²). The results of the case studies show that GSim has better execution efficiency for ABS modeled in continuous space.

A top-down cloning strategy to accelerate parameter space exploration on GPU: Because of the importance of the parameter space exploration and its close relationship with the ABS, the efficient execution of multiple ABS instances on GPU is further investigated. In this research, an incremental simulation cloning technique is investigated. An ABS cloning strategy, named top-down cloning, is proposed and elaborated. The implementation of our proposed strategy on GPU supports concurrent execution of agents within each simulation instance as well as concurrent execution of multiple simulation instances. Performance of the proposed top-down cloning strategy is evaluated and analyzed. Our experiment results demonstrate that cloning can significantly speed up the overall parameter exploration task.

A bottom-up cloning strategy to accelerate parameter space exploration on GPU: The top-down cloning strategy can successfully achieve the goal of exploiting the computation sharing and shorten the overall execution time of a parameter space exploration task. However, it only allows one parameter difference between two simulation instances that share computations, and it requires agents encounter environment parameters in the same sequence. To overcome these limitations, a bottom-up cloning strategy is therefore proposed. The design of the bottom-up cloning strategy is elaborated. An evacuation scenario under different environment settings is adopted as a case study to demonstrate the effectiveness and generality of the bottom-up cloning strategy.

1.5 Structure of the thesis

The rest of the thesis is organized into the following chapters.

- Chapter 2 reviews the related work on agent-based simulation and parameter space exploration. For agent-based simulation of single instance, the topics include various ABS applications, ABS toolkits, techniques that aim to accelerate ABS on

²<http://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-agent-based-simulation-complex-systems/>

GPU. Two research issues, the design of agent management module and the design of agent interaction module, are also studied in the literature survey. For parameter space exploration, the topics include various parameter space exploration applications, and the techniques to accelerate the parameter space exploration task. Among all the techniques, we particularly focus on using simulation cloning techniques to accelerate the overall parameter space exploration task.

- In Chapter 3, the design and implementation details as well as evaluations of the two common modules of ABS on GPU are presented, namely agent management module and agent interaction module. In particular, we demonstrate how our design of the two modules takes the advantage of GPU and overcomes the limitations of GPU. The implementation details and evaluation results compared with FLAME, a well established GPU based ABS framework, are also provided.
- In Chapter 4, the design and evaluation of an ABS execution library that can be commonly adopted for ABS are presented. In particular, under the hood, the library is implemented with modules discussed in Chapter 3. The library exposes a set of APIs for users to build and accelerate their ABS application on GPU.
- In Chapter 5, the technical details of a simulation cloning strategy to accelerate parameter space exploration task on GPU, i.e., the top-down cloning strategy, is presented. In particular, the mechanisms of the strategy are elaborated. Besides, the data structures, implementation and evaluations are also provided.
- In Chapter 6, the technical details of an improved simulation cloning strategy, i.e., the bottom-up cloning strategy, is discussed. In particular, the limitations of the top-down cloning strategy are identified and the design of the bottom-up cloning strategy is elaborated. A large scale scenario is used in the case study to demonstrate the effectiveness of the bottom-up cloning strategy.
- Finally, Chapter 7 concludes the thesis and discusses several directions for the future work.

Chapter 2

Literature Survey

In this chapter, a comprehensive review on the related research fields is presented. First, strategies to accelerate the execution of a single ABS instance are studied. The related works using GPU as an accelerator in general to speed up various applications are reviewed to understand the features of GPU and the applications that GPU can support. The literature describing various ABS applications and toolkits is studied to summarize the opportunities and challenges of accelerating ABS with GPU. Then, the literature that discusses the relationship in multiple simulation instances of parameter space exploration tasks is also studied. Particularly, the simulation cloning techniques are investigated.

2.1 Using GPU as an accelerator in general

Hardware accelerators, such as GPU, are promising parallel platforms for high performance computing. GPU provides an inexpensive, highly parallel system to application developers. There has been growing research and industry interest in accelerating applications with GPU. Compared to CPU platforms, it has shown prevailing advantages to run computation intensive and data parallel applications [68, 75].

There are numerous applications accelerated by GPU. The key issue of utilizing GPU is to exploit the data parallelism of the application. Among all the simulation applications, those using cellular automata (CA) model typically exhibit the data parallelism and are suitable for parallel execution on GPU. For example, the spin model [39] is a CA model. In the spin model, each cell computes a state transition based on surrounding cells. It is computation intensive and is used to analyze phase transitions occurring in

statistical mechanics and many other systems including social networks, physical computer networks, and web page relationship on the world wide web [39]. Wende [105] developed a GPU solution to accelerate the calculation of state transition in spin model. Many more works about using GPU to accelerate CA based applications can be found in [31]. In general, as computations can be performed on cells in parallel, cellular automata are suitable to be executed on GPU.

When data parallelism is not obvious, it is necessary to design new data structures and algorithms to exploit the parallelism. For example, algorithms to process graph are usually serialized. Hong et al. [46] introduced a GPU implementation of the level synchronous breadth first searching (BFS) algorithm to explore a graph. The CPU version of the graph exploration uses a queue which serialize the processing. However, to accommodate the GPU programming, the authors changed the queue-based algorithm into an $O(n)$ array-based algorithm. The concurrent processing on the array can make the graph exploration performed in parallel.

Perumalla et al. [80, 82] introduced a method to simulate the vehicle movement on GPU by using a field based model. This model maps the real world road data onto a grid, with each cell in grid having the probability of turning either left/right or up/down. Cells are processed in parallel by GPU threads. Whenever there is a vehicle in a cell, its route can be calculated according to the turning probability of the cell.

Speeding up discrete event simulation is an important topic in parallel and distributed simulation area. The major challenge is to exploit the parallelism from the application. However, the local causality constraint limits the parallelism that can be achieved.

In [79], Perumalla studied efficient implementation of the PDES on GPU. The particular application is a diffusion simulation (e.g., heat transmission and gas diffusion) which is computational intensive. The author proposed a hybrid approach of time-driven and event-driven simulation. The clock advancement mechanism is the same as the event-driven approach, that is, the clock advances to the time stamp of the nearest future event. However, in each time advancement, multiple events are extracted and aligned to the same timestamp to be simulated concurrently, which is similar to the time-driven approach. Events in the same time slot are processed in parallel. The hybrid approach demonstrates good speedups with large problem size in both CPU and GPU implementations.

Park & Fishwick [76, 77] proposed generic data structures to process the parallel discrete event simulation (PDES). In order to ensure the data parallelism, the future event list (FEL) is decomposed into sub-FELs that are assigned to GPU threads for concurrent processing. Kunzmulti [58] proposed algorithms to sort and group events according to event types to further reduce the branch divergence in executing discrete event simulation on GPU.

In summary, based on the existing works of using GPU to accelerate a wide range of applications, the architectural and programming features of GPU can be identified, that is, (1) to take advantage of the GPU architecture, applications with higher data parallelism would be more beneficial; and (2) the data structures are usually organized into arrays, and the dynamic memory allocations are often avoided.

2.2 Execution strategy for a single ABS instance

ABS allows people to model their real-world systems of interest in ways that are neither possible nor readily accommodated using traditional modeling techniques, such as discrete event simulation. Research in ABS becomes popular, because there are problems that have not been adequately addressed by discrete event simulation, and not readily modeled using the existing simulation software, toolkits, or development environments [90]. ABS demonstrates the data parallelism due to the autonomy. However, the way agents interact is different from the applications discussed above. For example, because of the ability to move, the neighbors of an agent may change during simulation execution, which is different from the spin model. Besides, agents usually interact by directly accessing each other's state, which is different from message passing in PDES. Because of the needs of using ABS, and its difference from discrete event simulation, strategies particularly for ABS need to be looked into.

In this section, the typical ABS toolkits are reviewed. Two common modules of ABS, i.e., agent management module and agent interaction module are identified and studied. Some detailed issues related to agent management and agent interaction on GPU are particularly investigated.

2.2.1 Typical ABS toolkits

Since developing an ABS for complex system often involves domain experts who may not be familiar with programming, an agent-based modeling and simulation toolkit is necessary to simplify the task of developing the model and running the simulation. There are many existing ABS toolkits residing on various platforms. The majority of the existing ABS toolkits, such as SWARM [67], NetLogo [107], Repast [73], and MASON [61] run only on single CPU platforms.

Besides the traditional toolkits developed for general purpose applications, various ABS toolkits are developed for specific applications. For example, the MWGrid framework [23] utilizes distributed ABS to study medieval military logistics. The DIVAS framework [3] provides pluggable domain-specific modules and visualization modules for rapid development of simulations (an urban traffic simulation is used as a case study to demonstrate the usage of DIVAS in [3]).

There are a number of modules commonly supported by these toolkits, such as the agent management, agent interaction, visualization, and simulation result analysis. Among these modules, the agent management module and the agent interaction module are two fundamental ones. The agent management module maintains the agent list and manages agent creation and deletion. The agent interaction modules supports neighbor searching so that an agent can interact with its neighbors. These two modules are widely supported by many ABS toolkits. They are implemented differently in various simulation toolkits. For example, *ContextBuilder* and *ContinuousSpace* in Repast, and *Scheduler* and *Continuous2D* in MASON serve as agent management module and agent interaction module respectively.

One limitation of the existing ABS toolkits is the execution speed. The limitation on execution speed restricts the feasibility of executing large-scale ABS. Traditional toolkits fail to exploit the parallelism of ABS and are often based on highly serialized algorithms for agent processing. Parallel and distributed ABS toolkits are therefore proposed to address the issue of execution inefficiency.

Collier & North [21] demonstrated the Repast HPC implemented in the cluster environment. The HPC version of Repast considers the situation where one simulation instance is distributed across several nodes in the cluster. It implements a dynamic

discrete-event scheduler with conservative synchronization. The main issue considered here is the cross-process communication and synchronization. This work provides interfaces to handle the parallel event scheduling. In each schedule, only events with global minimum timestamp can be executed, and this global minimum timestamp is calculated with MPI's `MPI_Allreduce` function. The HPC Repast also provides interfaces to manage the context sharing. Whenever an agent needs to interact with other agents, the agent can access the `SharedContext` where local or non-local entities are managed.

FLAME [87] is a novel framework to support execution of generic ABS in parallel on GPU. It allows users to specify attributes and behaviors (i.e., function prototypes) of agents in an XML model file. An XML translator will transfer the XML model file into the optimized executable CUDA code. The FLAME users need to implement agent behavior functions, but they do not need to know many implementation details about GPU programming by using FLAME.

Silva et al. proposed an annotation technique to simplify the implementation of multi-agent systems on symmetric multiprocessors (SMP) [91]. SMP is a hardware architecture in which several processors share a single memory system and a single operating system instance. The proposed annotation technique allows modelers define model parallelization granularity and switch the execution between SMP and sequential machines.

There are some other novel strategies and techniques aiming at accelerating ABS with parallel and distributed architectures in the literature. For example, in [19], Terracotta, a JAVA-based middle-ware for distributed shared memory, is adopted to facilitate parallel simulation of ABS in multiple JVMs. In [112], a strategy is proposed to distribute the ABS workload all over the Internet so that the calculation can be done on off-the-shelf computers, smart phones and other Internet connected devices that are capable of rendering websites and executing JavaScripts.

2.2.2 Strategies to accelerate ABS with GPU

There are existing works to support ABS on GPU. In [1], Aaby et al. proposed a latency hiding strategy to simulate agent-based models on multi-GPU and multi-core clusters. An ABS of the CA model is split into blocks. Each block is padded with R layers containing information of agents in other blocks. Updates on R layers are computed locally, and

are synchronized with remote nodes in every R steps. Consequently, communication frequency is reduced by R times. The key contribution of this work is a strategy to map CA blocks to computing units. The blocks could be mapped to nodes in a clusters, and the R -layer strategy reduces the socket communication cost. Or, the blocks could be mapped to faster GPU shared memory, and the R -layers reduces the frequency to access slower GPU global memory. In general, the proposed strategy is applicable to hierarchical architectures in which there is a cost to communicate between two levels in the hierarchy.

In [70], Moser et al. accelerated the simulation execution of a cluster behavior model on multi-core machines, and on a GPU platform. The cluster behavior discussed in the paper has the following two properties: 1) each agent is able to interact only with other agents in the same cluster; and 2) the size of the cluster is unchanged, implying that agents cannot move to another cluster during the simulation.

Agents in the two applications above are stationary (non-moving). Models with stationary agents are suitable to be accelerated with GPU, because stationary agents can be easily mapped to GPU threads, and memory allocation of stationary agents also facilitate the coalesced memory access. So, the model can be easily decomposed and executed in parallel. Substantial performance gains were reported in both papers.

However, not all ABS models are spontaneously efficient to be executed on GPU. For example, agents can move in many ABS models, and because of the agent movement, threads need to process different sets of neighbors in each time step. Therefore, the coalesced memory access cannot be easily achieved and the performance of the GPU execution of the ABS is downgraded. Besides, the above two pieces of work do not address the issue of inefficiency of agent management on GPU (i.e., dynamic agent creation and deletion). Since agent management and agent interaction commonly exist in ABS, literatures particularly about these two modules will be discussed in the following subsections.

2.2.2.1 Agent management on GPU

Agent management module mainly handles agent creation and deletion which involves frequent dynamic memory allocations. Dynamic memory allocation on GPU is non-trivial

because it can be a severe performance bottleneck due to the global synchronization [106]. Besides, as reported in [92], allocation and deallocation performed by CUDA’s native memory allocator are prone to fault under a heavy load. Other than dynamically allocated memory space, the memory pool strategy could be adopted. In [76] and [77], discrete event simulation for queuing network is performed on GPU. A shared future event list is preallocated and split into segments. Each segment is owned by a thread for new event generation. One drawback is that the length of the future event list needs to accommodate the fastest event generation rate. This leads to a huge memory consumption. The design of future event list is further improved in [94]. Taking the event generation rate of each thread into consideration, it proposed a balancing strategy that computes a proper segment to place new events. It does not fix a thread to a specific segment. The strategy tries to fill all segments at similar rates. The size of future event list in this case just needs to accommodate the average event generation rate. FLAME [87] also supports agent generation on GPU. However, each agent is only allowed to generate no more than one agent.

Breaking big memory space into segments can incur imbalanced insertion which results in requiring much more memory than actually needed. If the segment size is fixed, the number of agents to be added is limited (e.g., in FLAME). The existing strategies are not efficient to handle this problem. So, agent management strategies need to be designed. In Chapter 3, a design for agent management based on memory preallocation is proposed. Compared with other preallocation approaches, our design does not break the memory into segments. All threads access the same piece of memory concurrently and atomically, and each thread can create as many agents as needed within the capacity of preallocated memory. Compared with CUDA native memory allocator which can be used to support dynamic agent creation, our approach is more efficient due to memory preallocation.

2.2.2.2 Agent interaction on GPU

Efficient agent interaction strategies on parallel architecture can be found in the literature. In [81], Perumalla and Aaby studied ABS of the CA model, and discussed the data parallelism of ABS on GPU. As discussed above, CA model usually has strong data parallelism and is suitable for GPU processing.

Agent interaction in the network environment often shares the same research issue as graph processing on GPU, which has been well studied. For example, simulation of information propagation over complex networks is performed on GPU in [53]. It adopts Medusa [114], which is a generic graph processing framework on GPU, to support agent interaction in network environment.

However, simulating the interactions of agents that can move is different from simulating the interactions of stationary agents because the neighbors of agents change due to agent movement. Therefore, the coalesced memory access to GPU global memory cannot be guaranteed. Moreover, neighbor searching need to be performed by every agent in every time step. Finding neighbors in continuous space is harder than finding neighbors in cell-based discrete environment or network environment, because the number of positions of neighbors in continuous space can be arbitrary.

Agent interaction in continuous space is similar to N -body simulation [2] in essence. Typical implementation of N -body simulation on GPU is studied in the particle systems consisting of moving particles [34, 38, 43]. As shown in [29], ABS can use the same implementation as adopted in particle system to perform neighbor interaction. To be specific, the implementation decomposes continuous space into cells. Agents (or particles) are sorted based on the IDs of the cell they are in. Then, agents in the cells far away from a subject agent can be directly filtered out. This can save some computations in neighbor searching. However, the difference between ABS and the particle system is that an agent has more complex states. As a result, sorting will incur more data movement.

The major overhead in the state-of-the-art N -body algorithm on GPU is caused by memory access. FLAME [87] uses GPU texture memory, which is read-only, and adopts better caching policy to achieve higher bandwidth. However, the utilization of shared memory, which has even higher bandwidth, is rarely studied in the literature. So, a potential area for research is to develop efficient strategies using shared memory on GPU to support agent interaction.

2.3 Execution strategy for a parameter space exploration task

Replication is initially proposed by von Neumann [101] decades ago to provide fault tolerance, and is realized by duplicating processes, data, transactions, and services. Later, it is used to improve throughput by placing replicas in the proximity where a service is needed [33] and to increase the concurrency of access by replicating databases [8]. In the area of simulation, replications are used to gain statistically meaningful simulation results [4, 32]. While replication refers to the execution of multiple simulation instances with the same set of parameter configurations, parameter exploration is used to generate multiple simulation instances with different parameter configurations. It is used to explore different design alternatives or to answer “what-if” questions [25].

2.3.1 Parameter space exploration

The parameter space exploration is required in various scientific and engineering problems. For example, an efficient exploration of configuration space is proposed in [88] to study a physical problem of the identification of metastable structures in condensed phase systems. Parameters are explored in structural design, such as the design of aircraft [93]. In the area of micro-architecture design, the parameter space exploration is adopted to facilitate the development of 3D integrate circuits (ICs) [109] and 3D cache [98].

Parameter space exploration with ABS is typically used in the areas of social analysis, marketing, and military. For example, in [110] an agent-based model is designed to investigate the role of parental relationships and intergenerational reproduction of cultural capital to understand the long-term professional success of an elite family line during the Ming and Qing dynasties in imperial China.

In the marketing area, ABS is used to explore “optimal” marketing strategies and to investigate the risks of financial markets [96]. Several attributes of a company are evaluated: price, quality, time, function, services, relationship among customers and brand image. The company’s decision depends on how to distribute investments among the seven attributes. Parameter space exploration is used to evaluate various combinations of investment distributions.

Decraene et al. [26] introduced a framework named complex adaptive system evolver (CASE) as an exercise to aid the decision making in military field. The framework has three components: model generator, simulation engine, and the evolutionary algorithm. In the case study, CASE is integrated with map-aware non-uniform automata (MANA) framework [59] to simulate different scenarios of military tactic and exploit the weakness of tactical plans.

2.3.2 Accelerating parameter space exploration task

Different methods have already been proposed to explore the parameter space of discrete models automatically. In the NetLogo platform [107], the *BehaviorSpace* tool allows to explore automatically and systematically the parameter space. The parameter space is a Cartesian product of values that each parameter can take. When there are many parameters and value choices, the parameter space becomes huge and systematic exploration becomes impossible because of the excessive execution time.

Many works in the literature aim to reduce the workload of the parameter space exploration tasks. Efforts have been made in two directions. One direction is to shrink the size of the parameter space being explored (which is discussed below). The other direction is to reduce the computation in individual simulation instances through simulation cloning (which will be discussed in Section 2.3.3).

Data farming is one way to avoid the exploration of the entire parameter space. Data farming prototypes and tools [47] start the exploration with a random sample space. In the later rounds of processing, the parameter space of interest is evolved with different initial conditions of sample space, finer gradations, or greater ranges. High performance execution platform on the cloud has been used to execute numerous simulation runs [57]. Practical use of data farming techniques has also been explored in the military field [26, 48].

Another way of shrinking the parameter space is to use evolutionary algorithms (EA). Frameworks such as ECJ [62] and CASE [26] have already been integrated with EA. An inverse simulation strategy has been proposed to reduce the iterations of EA in parameter exploration process in [95, 96], and its implementation in HPC clusters is discussed in

[110, 111]. The HPC implementations focus on dispatching simulation instances to multiple computing nodes for high throughput. However, the above-mentioned approaches for parameter space exploration all create a separate simulation instance for each parameter value combination and do not consider the relationship between the instances.

Parameter space exploration using GPU has been investigated in [58]. Although insights on multi-level parallelism, i.e., instance-level and event-level, are provided, the proposed strategies are not specifically designed for agent-based simulations.

2.3.3 Simulation cloning techniques

The second direction to reduce the workload of the parameter space exploration is to exploit the computation sharing among simulation instances with different parameter configurations of the same model, which is also known as the simulation cloning. Cloning of PDES has been well studied. In [50], an incremental cloning technique for PDES has been proposed. It allows exploration of multiple execution paths concurrently. The main idea of this work is to clone logical processes incrementally at a decision point. The concepts of virtual logical process and virtual message are proposed. In [49, 51], two particular issues, the scalability of the cloning technique and the issue of the in-time cloning, are further discussed. However, the cloning techniques for PDES cannot be directly applied to clone ABS, because LPs in PDES communicate through direct interactions by message-passing; whereas interaction in ABS is usually indirect through state sharing.

Pecher et al. [78] studied an efficient execution strategy to simulate vehicle trajectories in transportation simulations in order to estimate several possible future states of the traffic system. A new algorithm is proposed to allow a subject with various states to be simulated at the same time. However, the algorithm evaluates the state variation of a single subject agent (i.e., a vehicle). It cannot be directly applied to parameter exploration task in which multiple agents have different states to be evaluated.

Chen et al. [17, 16, 13] investigated an incremental cloning technique for HLA-based distributed simulation. Particularly, a scenario tree is proposed for defining the routing space of each cloned simulation instance to isolate the interactions among different cloned simulation instances that are running in the same simulation federation. With the routing

space correctly defined, using the data distribution management (DDM) service provided by HLA/RTI, a federate will be able to send messages to other federates belonging to the same cloned simulation instance. In our approach to be discussed later, a clone tree is proposed. But, it is constructed in a different way and used for a different purpose.

Chen et al. [15, 14] further discussed the incremental cloning mechanisms including the active cloning and passive cloning. A middle-ware is created to manage the communications among cloned and non-cloned federates. However, such techniques cannot be directly applied to clone ABS on GPU. Federates in HLA-based distributed simulation are few, loosely coupled and computation intensive compared to agents in ABS. Besides, the message passing communication approach in HLA is different from the state sharing in ABS. Therefore, creating a middle-ware to manage the mapping and interactions of cloned agents is inefficient. Cloning strategies specifically designed for ABS need to be proposed.

In recent works on simulation cloning, Hanai et al. [36, 37] proposed a cloning strategy specifically to reduce the repeated computations in traffic simulation. Analyzing large-scale traffic by simulation needs repeating execution many times with various patterns of scenarios or parameters. Such repeating execution is usually redundant because the changes in different scenarios may be small. For example, blocking one road segment only changes traffic of a small area. To increase the execution efficiency, The traffic simulation is developed as a PDES, in which each junction is modeled as an LP. Simulation cloning is performed when road conditions are changed. The LPs corresponding to the changed junctions are cloned. The strategy achieves significant performance improvement. This application is new, however, the techniques discussed in the paper are fundamentally based on cloning of PDES.

The term “agent cloning” was also used in the literature, however, with a different meaning. In [89], an approach is proposed to replicate working agents to improve the efficiency of the system. The agent cloning was used to solve the performance bottleneck in cases where agents cannot perform tasks due to insufficient local resources. The authors used agent cloning to migrate agent to remote hosts when the local resources are insufficient.

2.4 Summary

In summary, the data parallelism of ABS needs to be exploited for GPU to accelerate its execution. The agent management and agent interaction are common operations that are frequently performed in ABS. Thus, efficient GPU execution strategies proposed for agent management and agent interaction have decisive effect on the overall performance of the ABS execution. In the next Chapter, the strategies supporting agent management and agent interaction are discussed.

To accelerate the execution of multiple simulation instances in a parameter space exploration task, simulation cloning technique can be adopted. Cloning has been chosen, because there is a large amount of research that has been done in the first direction, especially by the data-farming community. Besides, the existing work on cloning of PDES and HLA has shown promising results. However, to the best of our knowledge the ABS cloning has not been fully analyzed in the literature. Therefore, ABS cloning is particularly analyzed in this thesis. The ABS cloning strategies will be discussed in Chapter 5 and Chapter 6.

Chapter 3

Supporting Common Modules of ABS on GPU

3.1 Overview

ABS is a bottom-up approach to analyze complex adaptive systems, such as crowd and traffic. ABS is usually performed in a time stepped manner. Each agent performs a Sense-Think-Act cycle at every time step [52]. Specifically, an agent assesses surrounding neighbors and environment in the *sensing* stage. It determines actions based on the neighborhood in the *thinking* stage. Then, it executes actions in the *acting* stage that may change the agent’s state and influence the surrounding neighbors.

ABS reveals system level properties through agent interactions. As discussed in Chapter 2, there are efficient strategies in the literature supporting agent interaction in cell-based discrete environment and network environment on GPU. However, agent interaction in the continuous space is not well supported. The agent interaction in continuous space is a computation and memory intensive process, because neighborhood relationship needs to be updated in every time step. Neighbor interaction in continuous space environment considers each agent as an individual. In applications such as flocking boids in which agent behavior is simple, neighbor interaction can take over 50% of the total execution time¹. Therefore, handling neighbor interactions efficiently has decisive effects on the overall execution time of ABS.

¹Result is obtained by profiling the flocking boid application provided in MASON toolkit, the simulation is configured to have 2000 agents. Each agent has ~ 30 neighbors.

Parallel platforms, such as GPU, provide an opportunity to accelerate the ABS execution. The autonomy of agent sheds light on parallel processing. But, GPU is not spontaneously efficient to support ABS. To take advantage of GPU and address the issues of using GPU to accelerate ABS, strategies need to be proposed to improve the execution efficiency of two common modules of ABS on GPU, namely the agent management module and agent interaction module. Our main contributions are: 1) an **AgentPool** data structure to support multiple agent types and the dynamic creation and deletion of agents; and 2) an agent interaction module which takes advantage of GPU memory hierarchy to accelerate the neighbor searching. Case studies are conducted to compare the proposed strategies with a well established ABS framework on GPU, namely FLAME [87].

The rest of this chapter is structured as follows: Section 3.2 discusses the challenges to support the agent-based simulation on the GPU platform. Section 3.3 explains the strategies in detail, including the effective utilization of the GPU memory hierarchy and the corresponding algorithm design. Section 3.4 provides an evaluation of our strategies by using a flocking-boids application as an example. The performance statistics are also given in this section. Finally, Section 3.5 summarizes the chapter.

3.2 Challenges of executing ABS on GPU

GPU can provide numerous threads to process agents with greater parallelism than CPU. Executing agent-based simulation seems to be straightforward on GPU. Because of the autonomy, agents can be mapped to GPU threads in one-to-one correspondence. An agent's Sense-Think-Act behaviors can be specified as GPU kernel functions triggered in every time step. GPU kernel functions are executed by GPU threads concurrently. Such concurrency is the source of speedup.

Although the autonomy of agent makes GPU a favorable choice to execute ABS in parallel, ABS is not spontaneously efficient on GPU due to some unique features of ABS. However, the inefficiency caused by the branch divergence, dynamic memory allocation and improper memory access pattern can reduce the benefits of adopting GPU, which are discussed below.

First, straightforward GPU implementation of ABS may not fully comply with the SIMT execution model. Agents of multiple types may have various behaviors and may not agree on the same execution path. This causes branch divergence. Different execution paths handled by threads within a warp are processed in a serialized manner, which reduces the parallelism.

Second, the memory usage during ABS is not the favorable case for GPU. For example, agent creation and deletion are common operations in various applications. However, dynamic memory allocation is not well supported on GPU. Neighbor searching is also necessary for many applications. However, neighbors of an agent could be situated anywhere in the global memory. Coalesced memory access is nearly impossible in the neighbor searching operation.

To address the issue of branch divergence caused by processing agents of different types and the dynamic agent creation and deletion, an **AgentPool** data structure and associated operations are specially designed. The **AgentPool** manipulates the pointers in a big trunk of preallocated GPU memory to realize the functionalities of agent creation and deletion. Customized for agent-based simulation, the **AgentPool** is more efficient than performing **new/delete** operations with GPU threads according to our experiments in Section 3.4. Besides, agents of the same type are also aggregated in one **AgentPool** for simultaneous processing to accommodate the SIMT execution model, because agents of the same type have the same execution path.

To address the issue of non-coalesced memory access caused by performing agent interaction, instead of attempting to re-align agent data and to achieve coalesced global memory access, data of an agent's neighbors can be stored in the small but fast shared memory in a batch-by-batch manner. Bank conflict is also taken into consideration and minimized when the shared memory is utilized.

3.3 Strategy implementation

In this section, the design and implementation details of the agent management module and agent interaction module are described. An **AgentPool** data structure is designed to handle the agent management. For agent interaction, an enhanced neighbor sharing strategy is developed by efficiently utilizing the GPU shared memory.

3.3.1 Agent management

The **AgentPool** is proposed to aggregate the same type of agents and to perform operations of agent creation and deletion. Designed to be a template, one **AgentPool** instance accommodates agent of the same type. For a scenario with various types of agents, multiple instances of the **AgentPool** could be created. Aggregating agents of the same type in one **AgentPool** is to comply with the SIMT execution model of GPU. Grouping agents of the same type may not totally remove the branch divergence. However, the agents of different types have higher chances of carrying out different behaviors and incur more branch divergences. The **AgentPool** approach is a heuristic to reduce branch divergences caused by executing agents of different types. Besides, such design facilitate the calculation of memory space required by one **AgentPool**. An instance of the **AgentPool** is initiated with a fixed capacity, **numElemMax**, to accommodate **numElemMax** agents of the same type. At runtime, the actual number of agents in the pool is denoted as **numElem**. The total memory space of the **AgentPool** is allocated with **cudaMalloc** function provided in CUDA library. The **AgentPool** management is in general performed by manipulating pointers to realize agent creation and deletion to avoid operations with actual data, consequently to reduce the overheads.

Table 3.1: **AgentPool** arrays

Array	Sorted	Description
agentArray	No	Keep private data.
dataArray	No	Keep data shared with other agents: one for reading and the other for writing.
dataCopyArray	No	
dataIndexArray	Yes	Keep indices to data arrays.
takenFlagArray	Yes	Keep whether an slot is taken or not.

AgentPool contains several arrays as shown in Table 3.1. The relationship between these arrays is depicted in Figure 3.1. The first three arrays are data arrays that keep actual agent data. The **agentArray** is an array of **ConstantData**. The **ConstantData** keeps agent properties that do not change during the simulation (e.g., **agentID**, **maxSpeed**, and **sensingRadius**). In comparison, the **dataArray** and **dataCopyArray** are arrays of **RuntimeData**. The **RuntimeData** keeps agent properties that change as simulation proceeds (e.g., **location** and **velocity**). When agents are concurrently processed by GPU

Figure 3.1: Demonstration of AgentPool manipulation

threads, while an agent a_i updates its `RuntimeData`, other agents may also need to access a_i 's `RuntimeData`. To avoid potential write-and-read conflicts, two copies of `RuntimeData` are maintained in `dataArray` and `dataCopyArray` respectively, one for read and the other for write. At each time step, agents access their neighbors' data in `dataArray` and write their own updated status to `dataCopyArray`, so that the write-and-read conflicts are avoided. This is also known as the Ping-Pong mechanism. At the end of each time step, `dataArray` and `dataCopyArray` are swapped so that the updated agent properties can be directly accessed in the next step of processing. Elements in these arrays with the same index are owned by the same agent.

The other two arrays are auxiliary arrays to improve the efficiency of operations on the `AgentPool`. The `dataIndexArray` keeps the indices to `agentArray`. Initially, `dataIndexArray[i] = i`, for all i . The `takenFlagArray` is used to keep track whether a slot in the pool is taken or not. Initially, all slots are empty (i.e., `takenFlagArray[i] = false`, for all i). Elements in `dataIndexArray` have one-to-one correspondences with elements in `takenFlagArray`. After creation and deletion operations, agents will be aggregated to the front of the `AgentPool` based on the values in the `takenFlagArray`, as

Algorithm 1: AgentPool methods

```

1 Function remove( $AI_i$ ) {
2   |   takenFlagArray[ $AI_i$ ]=false;
3 }
4 Function safeSlot() {
5   |   return atomicInc(&incCount) + numElem;
6 }
7 Function add( $AI_i$ ) {
8   |    $DI_i$ =dataIndexArray[ $AI_i$ ];
9   |   takenFlagArray[ $AI_i$ ]=true;
10  |   return  $DI_i$ ;
11 }

```

shown in Figure 3.1(c). Agent states could be complex, so the size of the data arrays could be big. However, the auxiliary arrays are simply arrays of integers. Instead of directly performing aggregation operations on the data arrays, aggregating the two auxiliary arrays is more efficient. Agents' data can still be retrieved through `dataIndexArray` after the aggregation.

Each agent takes up a slot in the **AgentPool**. A slot is a composition of five elements, each taken from the above five arrays. The elements of agent a_i consist of elements in the two auxiliary arrays with index AI_i , and elements in the three data arrays with index DI_i , where

$$DI_i = \text{dataIndexArray}[AI_i].$$

AI_i is updated in every time step. DI_i remains unchanged. This means the memory space allocated to hosting a_i 's data is unchanged until the deletion of a_i .

The **AgentPool** provides two methods, namely **add** and **remove**, for agent creation and deletion (see Algorithm 1). As usual, due to the concurrency of GPU threads, creating or deleting agents should not influence the on-going processing. To ensure this, the deleted agents will be removed from the **AgentPool** only after all agents finish execution in the current time step, and the newly created agents will participate in the simulation only from the next time step.

Figure 3.1(b) and Figure 3.1(c) illustrate deleting agent with an **AgentPool**. The **remove** method takes an agent's index in the current time step, AI_i , as an input, and

returns nothing. It sets `takenFlagArray[AIi]` to be false. At the end of each time step, an aggregation operation is performed on `dataIndexArray` and `takenFlagArray` based on the values in the `takenFlagArray`. A GPU reduction operation on `takenFlagArray` could be performed to calculate the current total number of agents (i.e., `numElem`). Consequently, slots with index from `numElem` to `numElemMax` are vacant slots for later use. The corresponding slots in the agent data arrays are available for storing data for new agents.

Figure 3.1(d) illustrates adding agent to the `AgentPool`. The `add` method takes AI_i and returns DI_i for placing agent data. The key issue is to obtain a unique and safe AI_i for each newly created agent. Since multiple threads may create new agents concurrently during the simulation, the variable that keeps track the index of free slot may be accessed concurrently. To ensure safety, an auxiliary `safeSlot` method is proposed (see Algorithm 1). It returns `numElem + incCount` as the index of the next available slot, where `incCount` is increased atomically. When multiple threads request free slots at the same time, the atomic increment guarantees all of them will have non-conflicting slots. The `incCount` is set to zero at the beginning of each time step. After getting AI_i , the `add` method sets `takenFlagArray[AIi]` to true and `dataIndexArray[AIi]` to DI_i .

3.3.2 Agent interaction

The `Context` module is responsible for agent interaction in continuous space environment. The arrays in the `Context` are listed in Table 3.2. The `agentPtrArray` is a collection of references to all agents at the current time step. An `AgentPool` contains only the same type of agents. In an ABS, there may exist different types of agents. So, the `Context` may have agents from different `AgentPools`. `AgentPools` may change because of adding or deleting agents in each time step. Thus, an `AgentPool` needs to register its agents to the `Context` in every time step so that all types of agents can be found and interact with each other in the `Context`. To perform the registration, in each time step, the elements of `dataIndexArray` in an `AgentPool` need to be copied to `agentPtrArray` of the `Context`. Similarly, sorting `agentPtrArray` in `Context` does not require the movement of the actual agent data. This is different from the work reported in [29, 34], in which the actual data are moved.

The $O(n^2)$ time complexity is inevitable to perform neighbor searching with individual agents. The state-of-the-art algorithm reduces the total number of queries by decomposing continuous space into a grid of cells. Agents in cells that have no intersection with sensing radius will be ignored. Figure 3.2 illustrates the mapping of agents in a 2D continuous space to a 1D agent array. The id of the cell (i.e., CI) where agent resides is determined simultaneously by all the threads according to the agents' positions. The **agentPtrArray** is then sorted based on the cell ids.

To extract the agents of a particular cell, two extra auxiliary arrays, **cidStartArray** and **cidEndArray**, are used. The **cidStartArray**[i] records the index of the first agent in the sorted **agentPtrArray** residing in cell i (denoted as SI_i), and **cidEndArray**[i] records the index of the last agent (denoted as EI_i). By iterating from SI_i to EI_i , all the agents of the i^{th} cell could be identified. The **cidStartArray** and **cidEndArray** can be generated in parallel. They are initialized with -1 . Thread i processing agent a_i in the sorted **agentPtrArray** will compare its own cell id (i.e., CI_i) with the cell id of the next agent a_j in the sorted **agentArray** (i.e., CI_j), where $j = (i + 1) \bmod N_{all}$, N_{all} is the total number of agents in the simulation. If CI_i is not equal to CI_j , thread sets **cidStartArray**[CI_j]= j and **cidEndArray**[CI_i]= i , where $i, j \in [1 \dots N_{all}]$, as shown in Figure 3.2. Otherwise, a thread does nothing.

Because of the agent movement, updating **agentPtrArray**, **cidStartArray** and **cidEndArray** needs to be computed at every time step, and this cost cannot be avoided. However, the computation is done in parallel with highly efficient GPU algorithms. As shown in the experimental results, the cost is quite minimal compared to the overall simulation.

Table 3.2: Context arrays

Array	Description
agentPtrArray	Keep references to agent data. It is sorted according to cell id.
cidStartArray	Given a cell id CI_i , the agent in that cell starts at index cidStartArray [CI_i] in the sorted agentPtrArray .
cidEndArray	Given a cell id CI_i , the agent in that cell ends at index cidEndArray [CI_i] in the sorted agentPtrArray .

Given agent a_i 's position and its sensing radius R_i , a $Rect_i(u_i, l_i, d_i, r_i)$ can be computed. $Rect_i(u_i, l_i, d_i, r_i)$ is defined as the minimal rectangle formed by cells that overlaps

Figure 3.2: The basic neighbor searching strategy on GPU

with the agent sensing radius, as shown in Figure 3.2. Pair (u, l) represents the 2D coordinate of the upper-left cell in the rectangle, and similarly pair (d, r) represents the 2D coordinate of the lower-right cell in the rectangle. Neighbors are determined in a two-step filtering process. First, agents in $Rect_i$ are the candidates of a_i 's neighbors, and those outside $Rect_i$ are filtered. Second, agents within the $Rect_i$ are iteratively checked, and only those inside R_i are actual neighbors. Some collective values (e.g., total number of neighbors, and average speed of neighbors) can be accumulated at the same time. These collective values then can be used by the subject agent to determine actions to be performed.

The above procedure is summarized in Algorithm 2. Lines 1 and 2 indicate that each agent is processed by a GPU thread. Lines 3 to 14 indicate the processing is conducted cell by cell in $Rect_i$. In detail, Line 3 takes a cell of $Rect_i$. Line 4 converts the cell's xy-coordinate to an integer cell id (CI). Lines 5 and 6 compute the SI and EI . Given SI and EI , Lines 7 to 13 iterate all neighbors in the cell to perform agent interaction. Once the for-loop is over, the algorithm goes back to Line 3 to take the next cell of $Rect_i$ to process, until all cells of $Rect_i$ are processed.

The above strategy can well fulfill the neighbor searching on GPU. However, it is not the optimized solution in terms of performance. The motivation to execute agent-based

Algorithm 2: Agent interaction

```
input : agentPtrArray, cidStartArray, cidEndArray,  $R_i$ ,  $Rect_i(u_i, l_i, d_i, r_i)$ 
1 for  $i = 1$  to  $N_{all}$  do in parallel
2    $agent = agentPtrArray[i]$ ;
3   for  $cell(x, y)$  of  $Rect_i$  do
4      $CI_i = calcCellId(x, y)$ ;
5      $SI_i = cidStartArray[CI_i]$ ;
6      $EI_i = cidEndArray[CI_i]$ ;
7     for  $j = SI_i$  to  $EI_i$  do
8        $neighbor = agentPtrArray[j]$ ;
9       if  $isNeighbor(agent, neighbor, R_i)$  then
10         $updateCollectiveValues()$ ;
11         $agent.doSomethingWith(neighbor)$ ;
12      end
13    end
14  end
15   $agent.doSomethingWith(collectiveValue)$ ;
16 end
```

simulations on GPU is to take advantage of the massively large numbers of GPU threads to speed up the execution. Since neighbor searching is a memory intensive task, and the way described above to access the GPU global memory does not comply with the coalesced access, this simple strategy of using only the global memory is inefficient.

The following two strategies adopt the much faster but much smaller sized GPU shared memory. To better utilize the size limited shared memory, common neighbors of multiple agents are identified. The neighbor data are loaded to the shared memory and shared amongst multiple agents. Two strategies are discussed in following subsections to utilize the shared memory for neighbor sharing, namely the naive neighbor sharing (NNS) strategy and the enhanced neighbor sharing (ENS) strategy.

3.3.2.1 The naive neighbor sharing strategy

The neighbor searching task seems to be an ideal case to explore the size limited but fast shared memory because each agent is limited to interact with only a few others. It is hoped that these interactions are as fast as possible.

The shared memory of a GPU block is accessible by the threads in the same block. Since each thread processes one agent, the heuristic is to use one block of threads to

Figure 3.3: The naive neighbor sharing strategy

process a segment of agents in the sorted `agentPtrArray`. As agents are grouped based on their cell ids in the sorted `agentPtrArray`, it is very likely that a segment of agents in the sorted `agentPtrArray` are potential neighbors in the environment (i.e., the continuous space). Therefore, if these agents are put into the shared memory, they can be accessed directly by the threads in the same block. This can be the source of speeding up the neighbor searching process.

Suppose the shared memory per block can accommodate N_{smem} agents. Because of its limited size, $N_{smem} \ll N_{all}$. The number of threads per GPU block is N_{block} , (this is, N_{block} agents are processed by one block). N_{block} should be configured to be less than or equal to N_{smem} so that agents processed by one block can be fit in the shared memory. Threads cannot access shared memory of different blocks. Thus, for those potential neighbors who are not put inside the same shared memory, they will be fetched from the global memory. For example, in Figure 3.3, assuming $N_{smem} = N_{block} = 3$, every three agents in the sorted `agentPtrArray` are processed by the threads belonging to the same block. To determine its neighbors, agent 1 needs to access all the agents in its $Rect_1$. Agents 4 and 7 are within the same block of Agent 1, so they can be accessed by reading the shared memory. While Agents 0, 3, and 6 reside in the shared memory of another block, they have to be accessed through the global memory.

However, this approach does not take much advantage of the shared memory. The main issue is that the shared memory is too small to accommodate many neighbors. In

Suppose agents 1, 4, and 7 are processed by three threads of the same GPU block. Neighbors that need to be accessed by agents 1, 4 and 7 are in Rect₁, Rect₄ and Rect₇ respectively. All agents are located at GPU global memory in the beginning. Agents in the bounding box of Rect₁, Rect₄ and Rect₇ (i.e., Rect_{block}) can be loaded into shared memory for faster access speed. Since the shared memory is size limited, the agents can be iteratively loaded, processed then removed, until all agents in Rect_{block} are processed.

Figure 3.4: The enhanced neighbor sharing strategy

our case study, it is often the case that one agent has many neighbors, but only a very small portion of them could be loaded into the shared memory. Besides, the way to access the shared memory in this approach is not optimal because there are potentially a large number of bank conflicts (e.g., in the example shown in Figure 3.3, threads for Agents 1, 4, and 7 may access different locations of the same memory bank concurrently). Thus, the performance is highly dependent on the way the agents' data are mapped to the shared memory. It is also possible that the same agent data in the global memory may be accessed multiple times. For example, in Figure 3.3, the thread for Agent 1 needs to get data for Agents 0, 3, and 6 from the global memory. Similarly, the thread for Agent 4 also needs to access the same data.

3.3.2.2 The enhanced neighbor sharing strategy

Since the main issue is the poor utilization of the shared memory, our improved strategy tries to increase the shared memory usage. As mentioned above, because of the sorting, agents of the same cells or neighboring cells, which are potential neighbors to each other, are highly likely to be assigned to the same thread block.

Due to different locations, the $Rect_i$ of agents processed by the threads in the same block could be different, as shown in Figure 3.4. In the enhanced neighbor sharing strategy, instead of statically loading the data of a block of agents into the shared memory, dynamically caching the data of all the agents in a bigger rectangle $Rect_{block}(u_b, l_b, d_b, r_b)$ in the shared memory is attempted. $Rect_{block}$ is defined as the bounding box that encompasses $Rect_i (u_i, l_i, d_i, r_i)$ of the agents that belong to the same block, where $i \in [1, \dots, N_{block}]$. Four indices u_b , l_b , d_b , and r_b are defined respectively by $\min(u_i)$, $\min(l_i)$, $\max(d_i)$, and $\max(r_i)$, where $i \in [1, \dots, N_{block}]$, as shown in Figure 3.4.

In the enhanced neighbor sharing strategy, the data of agents inside $Rect_{block}$ are put into the shared memory batch by batch. Assume the total number of agents inside $Rect_{block}$ is N_{Rectb} , and the shared memory of one block can only accommodate N_{smem} agents at most, where $N_{smem} \ll N_{Rectb}$. The N_{Rectb} agents are divided into several batches and each batch contains only N_{smem} agents. Each time only one batch of N_{smem} agents is put into the shared memory for processing. After a batch is processed, the shared memory is refilled with the next batch until all N_{Rectb} agents are processed. Since agents's data are loaded into the shared memory, N_{Rectb} is not required to be equal to or less than N_{smem} , and there is also no constraint on N_{block} either.

When shared memory is organized in such a way, all the threads are regulated to start reading from the first agent of a batch concurrently before moving to the next agent. By doing so, all threads read the same shared memory location. The bank conflict is therefore avoided. However, this approach may incur unnecessary operations for some threads to access agents that are out of the sensing radius of interests. For example, in Figure 3.4, the thread for agent 1 needs to access Agents 2, 5, and 8 for the reason that these agents are inside the $Rect_{block}$ even though they are not inside $Rect_1$. The unnecessary operations to access irrelevant agents will be minimized when most areas in $Rect_{block}$ are common areas amongst $Rect_i$. For example, considering the block of threads for Agents 0, 3, and 6, the $Rect_{block}$ is exactly the same as $Rect_i$ ($i \in 0, 3, 6$). This can be possibly achieved by adopting an efficient spatial indexing strategy. In reality, since threads execute the same operation in parallel, even though the operation may not be necessary for some threads, processing them may not actually increase the total execution time.

The above strategy could be further optimized. As mentioned in Section 3.2, each thread block is divided into multiple warps, each of which consists of 32 threads. Rather

Algorithm 3: Enhanced neighbor searching execution

```

input :  $R_i, Rect_{warp}$ 
1 for  $warps$  do in parallel
2   repeat
3     load data of the next batch of neighbor candidates in  $Rect_{warp}$  from global
       memory to shared memory;
4     for  $agent$  processed by a warp do in parallel
5       foreach  $neighbor$  in shared memory do
6         if  $isNeighbor(agent, neighbor, R_i)$  then
7            $agent.updateCollectiveValues()$ ;
8            $agent.doSomethingWith(neighbor)$ ;
9         end
10      end
11    end
12  until scan through all neighbor candidates in  $Rect_{warp}$ ;
13 end

14 for  $agent_i, i \in [1 \dots N_{all}]$  do in parallel
15    $agent_i.doSomethingWith(collectiveValue)$ ;
16 end
    
```

than constructing $Rect_{block}$ for one block, the $Rect_{warp}$ of each warp could be constructed instead. By doing so, instead of accessing a bigger $Rect_{block}$, each thread in a warp could access its corresponding smaller $Rect_{warp}$. Hence, queries to the cells inside $Rect_{block}$ but outside the $Rect_{warp}$ will not be performed. Taking Figure 3.4 as an example, if the warp size is 1 and the block size is 3, $Rect_{warp}$ corresponds to $Rect_i$ of each agent. For Agents 1 and 7, using $Rect_{warp}$ instead of $Rect_{block}$ will save the queries to Agents 2, 5, and 8 and Agents 0, 3, and 6 respectively. A warp is chosen as the minimal unit to construct the bounding box because threads of a warp execute the same instruction concurrently. In this way, threads in a warp will also access the same location in the shared memory. Thus, the bank conflicts are avoided.

To accommodate the agents processed by each warp, the shared memory per block SM_{block} is split into several pieces of shared memory per warp (SM_{warp}). Assume the number of warps per block is N_{wpb} , then SM_{warp} can accommodate N_{smw} agents at most, where $N_{smw} = N_{smem}/N_{wpb}$.

The overall algorithm of the ENS strategy is shown in Algorithm 3. The logic of

Table 3.3: Experiment setups

<i>Subsection 3.4.1: Agent management</i> Scenario: synthetic scenario 1 (in each time step, agent carries out no behavior, other than destroying itself and then creating a new one) Number of agents: 8192; Number of agents deleted and then added: 0, 256, 1024, 4096, 8192.
<i>Subsection 3.4.2: Agent interaction</i> Scenario: Flocking Boids (compare three memory access strategies); Number of agents: 4096; Environment dimension: 1024*1024; Query radius: 150.
<i>Subsection 3.4.3: Comparison of the proposed modules with FLAME</i> Scenario: synthetic scenario 2 (in each time step, agent carries out no behavior, other than accessing neighbor in a give query radius) Number of agents: 16384; Environment dimation: 128*128; Query radius: 1, 4, 16, 64.

Algorithm 3 resembles that of Algorithm 2 in general, except that Algorithm 3 has a procedure of caching data in the shared memory. Line 3 includes the operations of fetching agents from the global memory and placing them into the shared memory, which has been discussed in detail in this subsection. Lines 5 to 10 show how the neighbors in the shared memory are loaded and processed, similarly to Algorithm 2.

3.4 Experiments

In this section, the performance of the agent management module and the agent interaction module is evaluated. To isolate impacts caused by our strategies from those caused by complex agent behaviors, synthetic scenarios with minimum agent behaviors are created for testing of the two modules respectively. Table 3.3 shows the setups for each scenario.

The experiments were conducted on a GPU platform, i.e., a Fermi-based NVIDIA Quadro 2000 GPU @ 1.25GHz with 1024 MB of main memory and 192 cores distributed over 4 SMs. This Fermi-based GPU has 48 KB shared memory. All SMs share an L2 cache of 768 KB. The CUDA programs were compiled with nvcc v6.0 and `-O3 -arch=sm_20` flags.

Our strategies was compared with FLAME [87]. FLAME is a novel framework to support generic ABS in parallel on GPU. It allows users to specify properties and methods

of agents within an XML file. An XML translator will transfer the XML model file into optimized executable CUDA code. For the ease of use, FLAME exposes interfaces for model developers to specify the application dependent agent behaviors, and hide the general mechanism in the background processing. For example, it performs many memory loads and stores between different layers of GPU memory hierarchies, and transforms the data between struct-of-array (SOA) and array-of-struct (AOS) form. The reason to perform such memory operations and transformation is to balance the efficiency and programmability. Data in SOA form can be managed with GPU texture memory. Data in AOS form are more favorable in terms of programmability.

3.4.1 Agent management

One of our main contributions is the design of the **AgentPool** to support efficient agent creation and deletion. Synthetic scenario 1 is to evaluate the efficiency of the **AgentPool**. In this scenario, a thread simply removes agent from **AgentPool** then adds a new one back. The agent has no behavior. Given 8192 agents in total, the percentage of agents to be removed and then added were varied from 0% to 100%. The 0% case is the baseline case where no agent is created or deleted.

Three agent creation and deletion mechanisms were compared, i.e., **AgentPool** mechanism, CUDA's **new/delete** (denoted as NativeAlloc), and FLAME's mechanism. Although the agent performs no behavior in the baseline condition, each step still takes time to finish because of the processing required by application independent mechanisms. Tests on the **AgentPool** and CUDA memory allocator were manually implemented, thus they process the simulation in the same way except the agent creation and deletion. FLAME, on the other hand, has its own processing steps generated by its XML translator. To measure the execution speed, average frame-per-second (FPS) is used as a metric, computed as *number_of_time_steps* divided by *overall_processing_time*. Performance downgrading (PerfDown) is used to measure the impact caused by agent creation and deletion. It is computed as

$$PerfDown_{strategy} = \frac{FPS_{base} - FPS_{strategy}}{FPS_{base}} \times 100\%.$$

As shown in Figure 3.5, the **AgentPool** and CUDA memory allocator have comparable execution speed in baseline condition case because of the same implementation.

Figure 3.5: Performance comparison on memory management

However, the performance downgrading of CUDA memory allocator increases rapidly as the number of new agents increases. Poor scalability makes this strategy unsuitable for models involving frequent and massive agent reproduction and mortality. However, even with 100% new agents, the **AgentPool** incurs only a small overhead in comparison with the baseline condition case. The performance downgrading is only about 7.66%. FLAME is implemented differently and has higher execution speed than other two approaches. The performance downgrading of FLAME is also smaller, which is about 3%. However, FLAME achieves this by allowing each thread to create at most one new agent at a designated position in a preallocated agent array at each time step. This limits the type of applications that the FLAME can support (e.g., application in which agent has possibilities of generating more than one new agents). On the other hand, our proposed **AgentPool** does not have this limitation as long as the preallocated memory space for the **AgentPool** is large enough.

3.4.2 Agent interaction

3.4.2.1 Impact of memory access patterns

The model used in this test case is the Flocking Boids model with 8192 agents. As discussed in Section 3.3, the memory access pattern is crucial to perform agent interaction. To understand the impact of global vs. shared memory access, the performance of three

neighbor accessing strategies introduced in the last section were profiled using NVIDIA NSight Profiler v4.5, i.e., the pure global memory (PGM) strategy, the naive neighbor sharing (NNS) strategy, and the enhanced neighbor sharing (ENS) strategy. In the PGM, all data access is through global memory. In the NNS and ENS, part of the data are loaded into shared memory. The differences between these two approaches are the shared memory access patterns and the proportion of data loaded into the shared memory.

The profiling results of different memory access patterns are shown in Table 3.4. Several metrics are considered: the total number of requests to global and shared memory (i.e., GM Req and SM Req), and the bank conflicts per request (BC/Req). All metrics are accumulated for all agents in every time step.

Table 3.4: Profiling neighbor searching strategies

	GM Req		SM Req		BC/Req
	Load	Store	Load	Store	
PGM	0.71M	1.02K	0	0	0
NNS	0.56M	1.02K	0.26M	3.1K	0.48
ENS	0.34M	1.02K	0.51M	141.9K	0.06

Regarding the metric of GM Req, the NNS and ENS have fewer global memory load requests due to the use of the shared memory. Since ENS puts more data in shared memory, it has even fewer global memory access than NNS. All strategies have the same number of global memory store requests because in all three cases, the updated states of agents write back to the global memory. As for the metric of SM Req, PGM does not use the shared memory. So, it has zero load and store for SM request. Compared with NNS, ENS stores more data in the shared memory and also reads the shared memory more often. Table 3.4 shows that ENS accesses the shared memory more often than the global memory; whereas NNS is the other way round. This basically explains why ENS outperforms NNS. In addition, regarding the metric of BC/Req, as discussed in Section IV, the NNS does not address the issue of bank conflicts when using the shared memory. The ENS improves the memory access pattern by regulating threads of a warp to access the same address in the shared memory. Such access pattern forms a broadcast which is equal to one memory transaction [74]. Thus, ENS has the smaller BC/Req value than NNS.

Figure 3.6: Performance comparison among PGM, NNS and ENS

3.4.2.2 Performance comparison of strategies

Given the profiling results shown above, the following experiments further demonstrate how memory access patterns impact the performance of agent interaction. Figure 3.6 shows the execution speed and the relative speedup of each strategy. As the number of agents increases from 4K to 16K in a fixed size continuous space, the environment gets denser (i.e., each agent has more neighbors to interact with). The execution speed is measured by frames per second. The relative speedup is computed as the ratio of FPS of a subject strategy to FPS of the pure global memory implementation.

As shown in Figure 3.6(a), when the number of agents is relatively small, NNS performs the best. In this situation, each agent will have less number of neighbors on average. So, the rate for NNS to find a neighboring agent in the shared memory is high. In addition, the search range for ENS (i.e., $Rect_{block}$) is larger than that of NNS (i.e., $Rect_i$), resulting more computation overhead. Both of the above factors contribute to the good performance of NNS when the number of agents is relatively small. However, as the number of agents increases, the environment will get denser and each agent will have more number of neighbors on average. Since NNS copies only a fixed number of agents to the shared memory, in order to find all the neighbors its access to the global memory increases accordingly. This is not the case for ENS because it copies more number of agents to the shared memory when the environment becomes denser. As shown in

Figure 3.7: Comparison between ENS and FLAME in agent interaction

Figure 3.6(b), the performance benefit of ENS becomes rather obvious when the scale of simulation increases.

3.4.3 Comparison with FLAME

To compare the efficiency of agent interaction between our proposed approach and FLAME, the scenario is set up as follows: 16384 agents in total are distributed into a grid of 128 cells by 128 cells. Agents in the cells of the Von Neumann neighborhood of the cell where the subject agent situates are queried. Given a radius, the agent step function counts the number of neighbors covered by the querying range of the agent. As shown in Figure 3.7(a), by increasing the querying radius from 1 to 64, it is shown that the FLAME performs less efficient than our ENS technique. Its frame rate reduces much faster. Relative speedup is defined here as the ratio between the execution time of the FLAME and the execution time of the scenario when the ENS technique is used. Figure 3.7(b) shows that the relative speed increases rapidly. When the query radius is 64, FLAME is 9.12 times slower than the case using our ENS technique.

3.5 Summary

In this part, strategies to accelerate two common and fundamental modules of ABS on GPU are proposed to improve the performance of ABS, namely the agent management

module and agent interaction module. The agent management module supports agent creation and deletion. An **AgentPool** data structure and associated operations are designed to increase the efficiency of agent management. To improve the performance of the agent interaction module on GPU, an efficient neighbor searching execution strategy is proposed by carefully utilizing the GPU memory hierarchy, especially the shared memory. To demonstrate the performance of the proposed strategies, the **AgentPool** was compared with CUDA native memory allocator and FLAME to test dynamic agent creation and deletion, and the proposed agent interaction module was compared with FLAME. The experiments show that, when continuous space is used in the agent-based simulations, our implementation is able to achieve better performance than the FLAME on GPU.

Since the agent management module and agent interaction module are common in ABS, to ease the pain of reuse the proposed agent management module and agent interaction module, an ABS execution library is proposed, which encapsulates the implementation and optimization details and exposes a set of interfaces. In the next chapter, the library is described, and its effectiveness and generality is demonstrated by using it to implement several typical applications.

Chapter 4

A Library Supporting ABS on GPU

4.1 Overview

The agent management module and the agent interaction module commonly exist in various ABS frameworks. These two modules can be reused for various ABS applications. Besides, the execution speed also depends on these two common modules, especially when agent behaviors are simple. However, developing and optimizing these two modules from the scratch requires great efforts. So, it is desirable to have a programming library offering high programmability and performance to increase the productivity and efficiency of building and executing ABS.

GPU is suitable to accelerate ABS because agents can be intuitively mapped to GPU threads for parallel processing. However, running ABS on GPU requires the experience of GPU programming. Writing a correct and efficient GPU program is challenging in general, and it is even more difficult to build ABS applications on GPU. First, GPU supports thousands of concurrent threads. Developers need to write parallel programs that can scale to thousands of threads to fully utilize such parallelism. Different from CPU threads, GPU threads are executed in a lock step manner due to the SIMT parallel execution model. This feature requires reconsideration in algorithmic design for an ABS when there are multiple types of agents running different logics are assigned to GPU threads. Second, different from manipulation of memory on CPU, GPU has a unique memory hierarchy and different strategies should be applied to optimize each level in the memory hierarchy. The agent interaction module discussed in the previous chapter is to address the issue of inefficient memory access caused by irregular agent interactions.

However, it takes efforts to implement the proposed strategies for each ABS application. Finally, although CUDA programming and C/C++ programming have similarities, programming on GPU still needs to comply to a specific paradigm that requires learning. All these factors make developing and optimizing ABS applications on GPU a challenging task.

To ease the difficulty of using GPU, a library named *GSim* is proposed to simplify the process of building and executing ABS on GPU. GSim is inspired by MASON, a renowned ABS framework running on CPU. It aims to facilitate the building of ABS applications on GPU. Particularly, GSim provides built-in APIs for agent interaction and agent management. It also provides an ABS skeleton customizable for application specific agent behaviors. It is necessary to mention that agent behaviors are simultaneously performed by GPU threads in continuous space models.

GSim is designed with the consideration of efficiency and programmability. In the experiments, it is shown that in comparison with a CPU and a GPU framework for agent-based simulation, i.e., MASON and FLAME respectively, GSim performs better than MASON and FLAME in terms of execution speed.

The rest of this chapter is structured as follows. Section 4.2 describes the classes and their usage in GSim library. An example application, flocking boids, is used to demonstrate the workflow of GSim. Section 4.3 analyzes the usability of GSim by comparing it with FLAME. Section 4.4 demonstrates the performance comparison among GSim, MASON and FLAME. Finally, Section 4.5 summarizes this chapter.

4.2 GSim library

GSim is composed of a set of classes to build a simulation. In this section, the functionalities of these classes are described from GSim users' perspective. The design details of the classes have been discussed in Chapter 3. The workflow that connects all the classes to form the ABS is also described.

4.2.1 Classes

GSim provides simple classes with extensibility to accommodate various simulation scenarios. Three structures/classes, `ConstantData`, `RuntimeData` and `GModel`, need to

Figure 4.1: GSim classes

be customized to formulate application specific logics. Besides, another four classes, **AgentPool**, **Context**, **GRandom** and **GVisual**, provide APIs to realize the functionalities of agent management, agent interaction, random number generation and visualization. The relationship and important APIs of the classes are illustrated in Figure 4.1. The usage of these classes are described in the following subsections.

4.2.1.1 GModel

The **GModel** works as the controller of a simulation. Each **GModel** represents a simulation instance. It maintains multiple **AgentPools** hosting multiple types of agents and contains an agent interaction module (i.e., **Context**). It optionally contains a random number generator module (**GRandom**) and a visualization module (**GVisual**). The creation of **AgentPools**, **Context**, **GRandom** and **GVisual** is performed in the constructor of **GModel**.

GModel controls ABS execution with three functions: **start**, **step**, and **stop**. The way to customize these three functions is described as follows:

- The **start** function is for initialization. Users can initiate the **AgentPools** by creating initial agents and adding them to **AgentPools** in the **start** function.
- The **step** function is iteratively triggered to perform agent behaviors. It triggers the step methods of agents through the **stepPool** API of the **AgentPool**.

<pre>struct ConstantData { int id; double maxSpeed; double sensingRadius; };</pre>	<pre>struct RuntimeData { double2 location; double2 velocity; double2 goal; };</pre>
--	--

Figure 4.2: Example `ConstantData` and `RuntimeData`

- The `stop` function is called at the end of the simulation to finish jobs such as output simulation results to files.

4.2.1.2 AgentPool and ConstantData/RuntimeData

An instance of `AgentPool` class contains agents of the same type. Agent data is defined by the `ConstantData` and `RuntimeData` of agent. `ConstantData` is the data type of `agentArray` in `AgentPool`, and `RuntimeData` is the data type of `dataArray` and `dataCopyArray` in `AgentPool` (as discussed in Section 3.3.1). Therefore, as shown in Figure 4.1, `AgentPool` has one unit of `ConstantData` object and two units of `RuntimeData` objects. `AgentPool` is a template class. A user needs to define agent specific `ConstantData` and `RuntimeData` (e.g., Figure 4.2) and pass them as template parameters to `AgentPool`. An `AgentPool` instance is created in the following way:

```
AgentPool<ConstantData, RuntimeData> *pool  
    = new AgentPool<ConstantData, RuntimeData>(poolCapacity);
```

The `AgentPool` class provides `add` and `remove` methods to manage agent creation and deletion (as discussed in Section 3.3.1). Besides, it provides a `stepPool` method to trigger the agent step functions. The agent step function is agent specific, i.e., each type of agent can develop its Think-Sense-Act cycle and specifies it in its step function, as shown in Figure 4.3. The step function of agents in one `AgentPool` can be triggered concurrently with GPU threads. Aggregating and processing of agents of the same type is to comply to the GPU SIMT execution paradigm and avoid the branch divergence [35].

At the end of each time step, `dataArray` and `dataCopyArray` are swapped in the `AgentPool.swap` function (as shown in Figure 4.4) so that the updated agent properties can be accessed in the next round of processing. Hence, agent always reads `dataArray` and writes to `dataCopyArray`.

```
void stepPool(gModel) {  
    for i = 1 to numElem  
        stepAgent(gModel.context,  
            agentArray[i], dataArray[i]  
            dataCopyArray[i]);  
}
```

Figure 4.3: `AgentPool.stepPool` function

```
void swap() {  
    RuntimeData *temp = dataArray;  
    dataArray = dataCopyArray;  
    dataCopyArray = temp;  
}
```

Figure 4.4: The `AgentPool.swap` function

`AgentPool` also registers the agents in the pool to `Context` at the beginning of each simulation step (as discussed in Section 3.3.2). The registration is done with the `registerPool` method. GSim user should explicitly call `registerPool` of each `AgentPool` to register all agents to the same `Context`, so that agents of different types can interact with each other.

4.2.1.3 Context

`Context` provides APIs to conveniently and efficiently perform agent interactions. The strategy and GPU implementation have been discussed in Section 3.3.2. The functionalities of `Context` are encapsulated and exposed as two APIs, `initNeighbors` and `nextNeighbor`. The two APIs are called in agent step function to realize the agent interaction, as shown in Figure 4.5 (it is an example implementation of the `stepAgent` function in Figure 4.3). The `initNeighbors` takes the location and sensing range of agent a_i as parameter and returns a bounding box `rect` that encompasses agent a_i 's neighbors. The `nextNeighbor` takes `rect` as a parameter and returns the data of next neighbor one by one until all neighbors are iterated. The neighbors are fetched one by one instead of being returned as a collection in one time because constructing the collection of neighbors requires dynamic memory allocation, which is inefficient on GPU. The neighbor data is fetched from the GPU shared memory using the mechanism discussed in Section 3.3.2.

4.2.1.4 GRandom and GVisual

`GRandom` and `GVisual` are self-explanatory. `GRandom` wraps the CUDA `curand` library to provide random numbers. `GVisual` is a simple visualization module using `cuda_gl_interop`

```

void stepAgent(context, cdata, rdata, rdataCopy) {
    rect = context.initNeighbors(rdata.location, cdata.sensingRadius);
    neighbor = context.nextNeighbor(rect);
    while(neighbor  $\neq$  NULL) {
        performThink(rdataCopy, neighbor);
        neighbor = context.nextNeighbor(rect);
    }
    performAct(rdataCopy);
}

```

Figure 4.5: Using **Context**'s neighbor searching APIs to realize agent interaction

(CUDA and OpenGL interoperability) library to visualize agents location and movement in the continuous space represented by **Context**. The implementations of these two modules are based on the existing APIs of CUDA.

To summarize, classes provided by the GSim are listed in Table 4.1.

Table 4.1: Module descriptions

GModel	Represent ABS controller. Need to be customized.
AgentPool	Manage agent creation and deletion. Maintain RuntimeData and ConstantData .
Context	Handle agent interaction.
GRandom	Provide random numbers.
GVisual	Provide basic visualization.
RuntimeData	Represent changing agent properties. Need to be customized.
ConstantData	Represent constant agent properties. Need to be customized.

4.2.2 Workflow

The APIs provided by **AgentPool** and **Context** listed in Figure 4.1 can be directly utilized. The user just needs to implement the **start**, **step** and **stop** function of **GModel** class, and agent **step** function.

To demonstrate the usage of GSim APIs and the workflow of GSim Application, a sample flocking-boids application is written as shown in Figure 4.6. For simplicity of the demonstration, the example is written in C/C++ style codes because CUDA programming resembles C/C++ programming. The actual runnable code can be found

```

01 // include AgentPool, Context, GRandom, GVisual
02 #include "GSim.h"
03
04 typedef struct ConstantData {
05     int agent_id;
06 } Boid_C_Data;
07
08 typedef struct RuntimeData {
09     double2 position;
10     double2 velocity;
11     double2 goal;
12     // init properties randomly with GRandom
13     void init(GRandom *rand);
14 } Boid_R_Data;
15
16 void stepAgent(Context* context, Boid_C_Data* cdata,
17     Boid_R_Data* data, Boid_R_Data* dataCopy){
18     Rect rect = context.initNeighbors
19         (data.location, cdata.sensingRange);
20     Boid_R_Data *neighbor
21         = context.nextNeighbor(rect);
22     double2 cohesion, consistency, avoidance;
23     int numNeighbors = 0;
24     while (neighbor != NULL) {
25         cohesion += calcCohen(data, neighbor);
26         consistency += calcConsis(data, neighbor);
27         avoidance += calcAvoid(data, neighbor);
28         numNeighbors++;
29         neighbor = context.nextNeighbor(rect);
30     }
31     double2 newVelocity = (cohesion + consistency +
32         avoidance) / numNeighbors;
33     dataCopy->velocity = newVelocity;
34     dataCopy->location += newVelocity;
35     dataCopy->goal = updateGoal(dataCopy->location);
36 }
37
38 int main(int argc, char **argv)
39 {
40     #define NUM_STEP 1000
41     GModel *boidModel = new GModel();
42     boidModel->start();
43     for (int i = 0; i < NUM_STEP; i++)
44         boidModel->step();
45     boidModel->stop();
46 }
47
48 class GModel{
49 public:
50     AgentPool<Boid_C_Data, Boid_R_Data> *pool;
51     Context *context;
52     GVisual *visual;
53     GRandom *random;
54
55     GModel()
56     {
57         #define POOL_CAPACITY 128
58         pool = new AgentPool<ConstantData,
59             RuntimeData>(POOL_CAPACITY);
60         context = new Context(POOL_CAPACITY);
61         random = new GRandom();
62         visual = new GVisual();
63     }
64
65     void start(){
66         #define NUM_AGENT_INIT 32
67         for (int i = 0; i < NUM_AGENT_INIT; i++)
68             in parallel on GPU {
69                 int agentIndex = pool->safeSlot();
70                 int dataIndex = pool->add(agentIndex);
71                 // agentArray contains Boid_C_Data
72                 pool->agentArray[dataIndex].agent_id
73                     = dataIndex;
74                 // data(Copy)Array contains Boid_R_Data
75                 pool->dataArray[dataIndex].init(random);
76                 pool->dataCopyArray[dataIndex]
77                     = pool->dataArray[dataIndex];
78             }
79     }
80
81     void step(){
82         // copy agents in AgentPool to Context
83         pool->registerPool(context);
84         // step agents in AgentPool
85         pool->stepPool(this);
86         // swap dataArray and dataCopyArray
87         pool->swap();
88         // draw the all agents in Context
89         visual->draw(context);
90     }
91
92     void stop(){
93         // output simulation results to file
94     }
95 };

```

Figure 4.6: A flocking boid application implemented with GSim

on Github site¹.

In line 2, the `GSim.h` is included, which contains the definitions of `AgentPool`, `Context`, `GRandom`, and `GVisual`. In this simulation, there is only one type of agent called boid. In lines 4 to 14, the `ConstantData` and `RuntimeData` of a boid are customized and renamed as `Boid_C_Data` and `Boid_R_Data`.

Lines 16 to 33 define agent step function. the step functions are executed by multiple GPU threads concurrently. In line 18, `Context` initializes the neighbor searching task with the subject agent's location and sensing radius and generates a bounding box,

¹<https://github.com/xiaoyaoth/GSim>

`rect`, that encompasses all neighbors. Line 19 returns the first neighbor. The cohesion, consistency and avoidance between the current agent and its neighbors are computed in lines 22 to 28 until all neighbors are iterated. lines 29 to 32 update the agent states for the next time step.

lines 34 to 77 define the `GModel`. `GModel` of the boid simulation contains the following components: an `AgentPool` hosting the `constantData` and `RuntimeData` of boid agents, a `Context` instance, a `GVisual` instance and a `GRandom` instance. The initialization of `GModel` components is in lines 41 to 48. The `GModel.start` method adds initial agents using the `safeSlot` and `add` APIs of `AgentPool` (the detailed usage of the two APIs are discussed in Section 3.3.1). The `GModel.step` function first registers boids in `AgentPool` to `Context` (line 65) to enable the neighbor searching in agent step function. Agent step function is triggered concurrently by multiple threads in the `stepPool` function in line 67. The `dataArray` and `dataCopyArray` are swapped (line 69). Finally, the visualization modules draw agent according to their locations at the end of the agent step function (line 71). The `GModel.stop` is used if any results need to be output to file.

The execution order of `GModel` functions can be viewed in the `main` function in lines 78 to 86. In line 81, the `GModel` instance representing a simulation is created. In line 82, `GModel.start` is called once. Then in lines 82 and 83, the `GModel.step` function is iteratively triggered, and the number of iteration is defined in line 80. Finally in line 85, the `GModel.stop` is called to output the simulation results to file and finalize the simulation.

4.3 Analysis of usability

This section discusses the usability of `GSim` in comparison with `FLAME`, which is a well established toolkit aiming at facilitating ABS development and accelerating ABS execution on GPU. The performance comparison between `GSim` and `FLAME` is demonstrated in the next section.

4.3.1 FLAME workflow

Before comparing with `FLAME` in terms of the usability, the usage of `FLAME` is briefly described. `FLAME` tries to minimize the effort of developing GPU codes. The agent

properties and simulation model specification need to be written as XML according to the FLAME manual² (Figure 4.7). The XML file is translated into CUDA code with the tool provided by FLAME. An agent step function signature is generated as the translation result (lines 1 to 3 in Figure 4.8). But, the body of the function needs to be implemented by FLAME user to realize agent behaviors (lines 5 to 27 in Figure 4.8). To perform agent interaction, the neighbors' states are represented by `messages` and passed to agent step function as parameters (e.g., `xmachine_message_location_list` in Figure 4.8). The `message` in FLAME actually means the data that are shared with other agents. It should be noted that, similarly to the `Context` in GSim, FLAME also returns messages one by one to avoid the dynamic memory allocation on GPU. Finally, user also needs to provide an XML file containing initial values of all properties of all agents. An example is shown in Figure 4.9. The name of the XML file containing the initial states of all agents is passed to the FLAME program as a command line argument.

4.3.2 Usability comparison

It is usually hard to argue whether software is of high or low usability, even though there are thousands of metrics serving as the guidance. In this part, the usability is analyzed based on the comparison. GSim is designed based on the object-oriented programming. GSim hides the complexity of GPU programming with class encapsulation, i.e., APIs are exposed for modules that require complex GPU programming. FLAME application is built upon XML specification for the model and C/C++ style programming for agent step function. FLAME hides the GPU programming by letting the user write XML description of the model and translating it into GPU code. In general, both GSim and FLAME achieve the goal of reducing the tedious work of GPU programming.

Two types of inconvenience of FLAME are identified in comparison with GSim. First, the learning curve of FLAME is steeper than GSim. Since many codes are hidden in both of the two frameworks, it is worth of discussing how the encapsulation affects learning the usage of the tools. From the perspective of software engineering, GSim differs from FLAME because of the difference of black-box reusability versus white-box reusability [102]. In GSim, since all modules are self-contained and well separated, the users do not

²<http://www.flame.ac.uk/docs>

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="FLAMEGPU_vcproj.xslt"?>
<gpu:xmodel xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
  xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
  <name>Boids</name>
  <gpu:environment>
    <gpu:functionFiles>
      <file>functions.c</file>
    </gpu:functionFiles>
  </gpu:environment>
  <xagents>
    <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>
        ...
      </memory>
      <functions>
        <gpu:function>
          <name>agent_step</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>
      </functions>
      <states>
        <gpu:state><name>default</name></gpu:state>
        <initialState>default</initialState>
      </states>
      <gpu:type>continuous</gpu:type>
      <gpu:bufferSize>128</gpu:bufferSize>
    </gpu:xagent>
  </xagents>
  <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>
      </variables>
      <gpu:partitioningSpatial>
        <gpu:radius>0.1</gpu:radius>
        <gpu:xmin>-0.5</gpu:xmin>
        <gpu:xmax>0.5</gpu:xmax>
        <gpu:ymin>-0.5</gpu:ymin>
        <gpu:ymax>0.5</gpu:ymax>
        <gpu:zmin>-0.5</gpu:zmin>
        <gpu:zmax>0.5</gpu:zmax>
      </gpu:partitioningSpatial>
      <gpu:bufferSize>128</gpu:bufferSize>
    </gpu:message>
  </messages>
  <layers>
    <layer>
      <gpu:layerFunction>
        <name>agent_step</name>
      </gpu:layerFunction>
    </layer>
  </layers>
</gpu:xmodel>
```

Figure 4.7: A flocking boid application implemented with FLAME, XML file

```
01 __FLAME_GPU_FUNC__ int agent_step(xmachine_memory_Boid* xmemory,  
02                                 xmachine_message_location_list* location_messages,  
03                                 xmachine_message_location_PBM* partition_matrix)  
04 {  
05     //Agent position vector  
06     float3 agent_position = make_float3(xmemory->x, xmemory->y, xmemory->z);  
07  
08     //get first neighbor, location_message represents a neighbor  
09     xmachine_message_location* location_message = get_first_location_message(location_messages,  
10                                     partition_matrix, xmemory->x, xmemory->y, xmemory->z);  
11     int numNeighbor;  
12  
13     while(location_message)  
14     {  
15         numNeighbor++;  
16  
17         //neighbor positions  
18         float3 message_position = make_float3(location_message->x, location_message->y, location_message->z);  
19         agent_position += doSomethingWith(message_position);  
20         location_message = get_next_location_message(location_message, location_messages, partition_matrix);  
21     }  
22  
23     //Update agent structure  
24     xmemory->x = agent_position.x;  
25     xmemory->y = agent_position.y;  
26     xmemory->z = agent_position.z;  
27     return 0;  
28 }
```

Figure 4.8: A flocking boid application implemented with FLAME, agent function

need to understand the implementation details of the provided classes other than the usage of the APIs. In contrast, users need to understand the detailed workflow of the FLAME to write correct XML model file in order to generate correct function signatures. Because FLAME is not organized into modules, functions generated by translating XML file and agent functions written by FLAME user are tightly coupled. The user needs to understand the overall workflow to use FLAME correctly. Thus, there is a steep learning curve to use FLAME.

Second, FLAME model is less flexible to change than GSim. In GSim, a parameter such as the size of the environment, the maximum number of agents can be easily changed. The changes could be done either through editing the code or through command line arguments. In comparison, parameters in FLAME need to be fixed in the XML model file, and the parameters of the simulation are non-changeable once the application is compiled. To setup a different configuration, the code needs to be recompiled. Changing parameters like environment dimension in code is prone to fault, because, after the XML translation, these parameters are hardcoded in many places. The safest way is to change the XML model file then regenerate the code. Moreover, the initial number of agents and


```
<states>
  <itno>0</itno>
  <environment/>
  <xagent>
    <name>Boid</name> <id>0</id> <x>-0.12465215</x> <y>-0.2607972</y> <z>-0.39554715</z>
  </xagent>
  <xagent>
    <name>Boid</name> <id>1</id> <x>0.4107396</x> <y>0.09933436</y> <z>0.18894494</z>
  </xagent>
  <xagent>
    <name>Boid</name> <id>2</id> <x>-0.32704008</x> <y>0.20930803</y> <z>0.14571494</z>
  </xagent>
  ...
</states>
```

Figure 4.9: A flocking boid application implemented with FLAME, the init file

each of their initial state needs to be fixed in initialization XML file as shown in Figure 4.9 before the simulation starts.

4.4 Experiments

To demonstrate the efficiency and generality of GSim, a series of applications are considered, including game-of-life [22], flocking boids [85], prey-and-predator [54], and the crowd simulation using social force [83]. These applications have multiple agent types and sophisticated agent behaviors. To be specific, application of flocking boids (FB) has agent behaviors of cohesion, consistency, avoidance and randomness. Application of crowd simulation using social force (SF) has agent behaviors of computing total force with neighbor agents and computing force with obstacles in the environment. Application of prey-and-predator (PP) contains two type of agents, and has agent behaviors of alignment, avoidance, approaching and repelling. Application of game-of-life (GoL) has agent behavior of determining live or dead status based on status of neighbor agents. Table 4.2 shows the setups for each scenario.

The reason of using these four applications is to show that our proposed library can accommodate common characteristics of ABS. In [84], the authors used a simple model, StupidModel, to show the characteristics typically appeared in the ABS of real systems. These characteristics are listed in Table 4.3. ABS frameworks are evaluated to see if part or all of these characteristics are supported. The four case studies, i.e., game-of-life (GoL), flocking boids (FB), prey-and-predator (PP), and the crowd simulation using social force (SF), are listed in Table 4.3 to show the supported characteristics. In

Table 4.2: Experiment setups

Scenario: Flocking Boids (FB); Number of agents: 1024, 2048, 4096, 8192; Environment dimension: 128*128; flocking radius: 10.
Scenario: Social Force (SF); Number of agents: 1024, 2048, 4096, 8192; Environment dimension: 128*128; Agent interacting radius: 6; Max velocity: 3.
Scenario: Prey and Predator (PP); Number of agents: 1024+16, 2048+32, 4096+64, 8192+128; Environment dimension: 1024*1024; escape radius: 60; hunt radius: agent.mass+60; flocking radius: agent.mass+20.
Scenario: Game of Life (GoL); Number of agents: 4096, 16384, 65536, 262144; Environment dimension: 64*64, 128*128, 256*256, 512*512; Query radius: 1 (Von Neumann neighborhood).

general, GSim can be directly applied to or work with the existing CUDA libraries, e.g., `curand` for randomness and `cuda_gl_interop` (CUDA and OpenGL interoperability) for visualization, to support most of the characteristics. However, the monitoring of simulation with histogram graph and built-in output are not implemented. Currently they are not the concern of this work and can be added by the application developer.

The experiments were conducted on two platforms. The first one is the CPU platform which consists of an 8-core Intel Xeon E5-2670 processor @ 2.6 GHz with a 20 MB L3 cache. The second one is the GPU platform, i.e., a Fermi-based NVIDIA Quadro 2000 GPU whose specifications and setups are described in Section 3.4.

The focus of this part of evaluation is to show the generality and performance advantages of GSim over the state-of-the-art CPU and GPU frameworks (i.e., MASON and FLAME) in four applications. MASON is included in the comparison to show the performance advantages of GPU over CPU in terms of execution efficiency.

Both agent management module and agent interaction module facilitate the agent-based simulation on GPU, especially for agents' environment modeled in continuous space. The `AgentPool` module expands the application domains of our strategies. Mul-

Table 4.3: Characteristics of general ABS [84]

Characteristics	Case studies	Remark
Randomness	FB, PP, SF	Initial position and velocity calculation are randomized.
Sensing neighbor agent	GoL, FB, PP, SF	The sensing operation is performed using the agent interaction module.
Sensing environment	SF	Obstacles are placed in the environment for agent to interact with.
Stopping rule that cause execution to end	GoL, FB, PP, SF	Currently simulation stops at a give time step.
Multiple types of agents	PP	Prey and predator are two types of agents with different behaviors.
Randomized agent execution ordering	GoL, FB, PP, SF	This bias caused by sequential execution is no longer a concern because of the concurrent execution of agents
Mortality and reproduction	Synthetic scenario 1 in Section 3.4	This characteristic is demonstrated in Section 3.4.
Read input files	Not implemented	Application dependent and can be hard coded.
File output and min, max, or mean calculation	Not implemented	Application dependent and can be hard coded.
Display value, histogram, or graph at runtime	Not implemented	This is not the concern of the work.

multiple **AgentPools** are used to support applications with multiple agent types (e.g., prey-and-predator). The agent interaction module is designed to accelerate the simulation execution. Focused on the agent-based simulation in continuous space, GSim outperforms MASON and FLAME in FB, SF and PP applications.

The game-of-life application is included in the case studies to show that GSim could also be an option for CA applications modeled using discrete cell-based 2D environment. In the discrete environment, the agent locations are discrete values rather than continuous values. However, GSim is not particularly optimized for ABS modeled in discrete environment. In the application of GoL, GSim does not outperform FLAME which has module optimized for the discrete cell-based 2D environment.

For a fair comparison, all application dependent logics are kept the same among GSim, MASON and FLAME. The MASON and FLAME implementations were both developed by faithfully following their manuals. Only the interfaces exposed to model developers

Figure 4.10: Performance comparison on various applications

to implement application dependent logics were used. The application independent processing in MASON and FLAME was implemented by using the proper interfaces to the best of our knowledge. For example, MASON and FLAME expose interfaces optimized for applications modeled in the discrete cell-based environment. Accordingly, implementations with our approach were also optimized by using the ENS to perform the agent interaction.

The detailed performance diagrams are shown in Figure 4.10. It shows obvious speedups of GSim over FLAME and MASON for applications modeled in continuous space (FB, SF, PP), especially when the scale of simulation increases. The source of the speedup, comparing GSim with MASON, comes from concurrent processing of agents. The speedup of GSim over FLAME is due to the optimization in the memory access pattern of the agent interaction module. As the number of agent increases, given fixed environmental configurations each agent has more neighbors to interact with. The growing trend of speedups indicates that GSim is more effective than the state-of-the-art approach. For simulations in discrete space, both FLAME and MASON provide optimization for the discrete environment; thus the FLAME outperforms GSim in the game-of-life application greatly. It should be noted that GSim still outperforms MASON when the number of agents is large.

4.5 Summary

In this chapter, an ABS library on GPU named GSim is proposed. The main contribution is to provide a set of easy-to-use APIs to facilitate implementation of ABS on GPU. The detailed optimizations for complex GPU programming are hidden, so that the users can focus on developing agent behaviors with the help of our provided tools. In the experiments, it shows that a wide range of applications could be supported by GSim. Particularly for models in continuous space environment, our library achieves speedups for simulations modeled in continuous space in comparison with the state-of-the-art CPU and GPU library, that is, MASON and FLAME.

With the help of the library, the construction and execution of single ABS instance are facilitated. In the next chapter, strategies accelerating a group of simulation instances

that forms a parameter space exploration task are investigated. Inside the group, the individual simulation instance is constructed with the proposed library. The common computations that exist among multiple simulation instances are exploited and saved. In this way, the overall execution will be accelerated.

Chapter 5

A Top-down ABS Cloning Strategy

5.1 Overview

One of the main applications of simulations is “what-if” analysis. An ABS usually contains a set of tunable parameters. The parameters could either be agent properties (e.g., maximal speed and sensing radius) or environment properties (e.g., placement and size of obstacles). By varying these parameters, agents will act differently. This will lead to different simulation results. To carry out a parameter exploration task or to evaluate different design alternatives, multiple simulation instances need be launched, each evaluating a different parameter configuration. It is possible that different simulation instances may share some common execution paths. So, computation can be saved by processing the common execution paths shared by multiple simulation instances only once. Simulation cloning technique is therefore proposed. The performance gain of the cloning can be promising if common execution paths are of high ratio.

To clone parallel discrete event simulation (PDES), an incremental cloning strategy has been proposed in [50]. It has also been subsequently applied to clone HLA-based distributed simulation in [17]. The cloning in the existing strategies is triggered when the internal states of LPs in PDES or federates in HLA-based distributed simulation are changed. A summary of the existing work on simulation cloning can be found in Chapter 2.

Agents are situated in the environment, and the ABS can be used to evaluate various environmental settings. Cloning strategies for ABS have not been studied in the literature to the best of our knowledge. It is necessary to design an ABS cloning strategy to help

with the exploration of various environmental parameters. Cloning agent for evaluating the environment parameters is different from cloning LP in PDES for evaluating different states of an LP, because the value of agent properties are not directly changed, but are influenced indirectly by the variations of the environment properties. This leads to different designs of the cloning strategies. In this and next chapter, the incremental cloning strategies for ABS will be investigated.

In this chapter, an approach named the top-down cloning strategy for incrementally cloning ABS will be described. The top-down cloning starts the exploration with one simulation instance evaluating the default value choices of environmental parameters specified by the modeler. Whenever an agent senses a parameter with different value choices, simulation instances evaluating other value choices of the parameter are cloned from the default instance. Multiple copies of the agent are created and added to the cloned simulation instances. The processing results of non-cloned agents in the default simulation instance can be shared with the cloned simulation instances. A cloned simulation instance (in short, a clone) can be further cloned when the agent in it senses other configurable parameters. Clones can form a parent-child relationship based on the computation sharing. Suppose C and C' are two clones. If C' is cloned from C , then C' will reuse the processing results of C , and C is the parent clone and C' is the child clone. The parent-child clone relationship forms a tree structure called the cloning tree.

The reason of carrying out the cloning-based parameter exploration task for ABS on GPU is that ABS is suitable for parallel processing due to an agent's autonomy and self-containedness. Furthermore, with proper design of execution strategy, not only a single ABS instance but also multiple simulation clones can be executed in parallel on GPU. Therefore, the total execution time of the parameter space exploration task is expected to be reduced.

The rest of the chapter is structured as follows: Section 5.2 discusses the difference between the cloning of PDES and ABS. The proposed cloning strategy, the top-down cloning strategy, is particularly designed for ABS cloning. Its design will be elaborated. Section 5.3 describes the GPU implementation and optimization of the top-down cloning strategy. Section 5.4 demonstrates the correctness and effectiveness of the proposed cloning strategy with a case study of an evacuation scenario, in which simulation is cloned to evaluate different sizes of exits. Finally, Section 5.5 summarizes this chapter.

5.2 Cloning agent-based simulations

A simulation model inherently has a multitude of configurable parameters (e.g., dispatching rules in manufacturing simulation and environment settings in evacuation simulation). A *simulation scenario* is defined by a combination of parameter values. To explore the simulation parameter space, a *stand-alone simulation instance* can be created for each combination of parameter values. Alternatively, simulation cloning techniques can be used. A *cloned simulation instance* (in short, *clone*) is created when a decision point is reached. Simulation can be cloned in its entirety or incrementally. In *entire cloning*, all simulation components (e.g., toolsets in manufacturing simulation and agents in evacuation simulation) are cloned at one time; whereas in *incremental cloning*, simulation components are cloned incrementally when necessary. To maximum sharing of computation amongst clones, in this work only the incremental cloning is considered.

5.2.1 Characteristics of cloning agent-based simulations

Cloning is usually triggered under two circumstances, at decision point or through interaction. In PDES, a decision point represents the changing of an LP's state and the start of the LP's execution path divergence. An LP reaches a decision point and actively clone itself when certain condition is satisfied (e.g., queue length of a toolset is greater than a predefined threshold level as in manufacturing simulation). Other LPs are further passively cloned when they communicate with the cloned LP.

In ABS, agents are typically situated in an environment. An agent may move in the environment and its behavior can be influenced by environmental settings (e.g., gates, obstacles). So, in our proposed simulation cloning strategies, an agent reaches a decision point, evaluating a cloning decision, when a configurable environmental setting is within its sensing range. For example, in the case study in the later section, cloning is used to evaluate different combinations of gate sizes in an evacuation simulation. The size of each gate is a configurable parameter. The *active cloning* occurs when the distance between agent and the gate is within a certain predefined range. Since environment settings affect an agent's behavior (e.g., gate size affects agent's movement in evacuation simulation), the cloned agent and its original copy will have different states. A cloned simulation instance is also generated to keep the cloned agents.

Active cloning generates a cloned simulation instance and clones the agent; whereas passive cloning only clones the agent. The *passive cloning* in ABS takes place when agents that are not cloned interact with cloned agents. However, unlike direct interaction between LPs in PDES by message-passing, interaction in ABS is usually indirect through state sharing. Different from PDES where LPs are loosely coupled by communication links, agents in ABS can be tightly coupled if they are situated closely in the environment (this is especially true for evacuation simulation). Tight coupling may cause passive cloning progress rapidly and thus diminishing the benefit of incremental cloning (that is, sharing of computation).

Compared with the LPs in PDES, the number of agents in ABS is much larger. As introduced in Chapter 3, references to all the agents are kept in **Context**. In incremental cloning, agents in a cloned simulation instance may need to interact with agents that belong to different clones. So, in order to obtain the correct **Context** of a clone, an execution ordering of clones must be determined and an efficient mechanism for **Context** creation and update must be developed.

5.2.2 Incremental cloning of agent-based simulations

Our proposed mechanism for ABS cloning allows exploring the entire parameter space with one session of execution. Suppose a parameter exploration task has n_p parameters. The i^{th} parameter p_i has n_i value choices. Each simulation instance evaluates a parameter value combination denoted as $(v_1, v_2, \dots, v_{n_p})$, where v_i is value choice of the i^{th} parameter to be evaluated, $1 \leq v_i \leq n_i$. The first value choice ($v_i = 1$) is the default value choice of the i^{th} parameter.

When cloning strategies are adopted to evaluate the parameter value combinations, simulation instance (i.e., a clone) that evaluates the parameter value combination $(v_1, v_2, \dots, v_{n_p})$ is denoted as $C(v_1, v_2, \dots, v_{n_p})$. So, $C(1, 1, \dots, 1)$ is the simulation instance evaluating the default parameter value combination. Agent a_j may have multiple versions in different clones, where j is the agent id. Its version in clone $C(v_1, v_2, \dots, v_{n_p})$ is denoted as $a_j^{(v_1, v_2, \dots, v_{n_p})}$.

For simplicity of explanation, an example shown in Figure 5.1 is used to illustrate the cloning strategy. In this example, agents are initialized at one side of an environment

Figure 5.1: Cloning tree and computation sharing

and move to the other side through two gates. The sizes of the gates are the parameters p_1 and p_2 , each of which has two value choices. The influences of various gate sizes on agent movement are evaluated.

The exploration task evaluates four parameter combinations, (1,1), (2,1), (1,2) and (2,2), with four clones as shown in Figures 5.1(a)-5.1(d). The computations are saved by processing agents that have the same behavior in multiple clones only once. The processing is done by one clone, and the results are shared with other clones. An agent will be cloned if it behaves differently under different simulation instances. The state of cloned agent needs to be updated in the corresponding simulation instance. For example in Figures 5.1(a) and 5.1(b), agents a_1 and a_2 have different states in $C(1,1)$ and $C(2,1)$ because they have been influenced by the different value choices of p_1 in the two clones. Thus, $a_1^{(2,1)}$ and $a_2^{(2,1)}$ are cloned from $a_1^{(1,1)}$ and $a_2^{(1,1)}$, then added to $C(2,1)$. However, the behavior of agents a_3 and a_4 are not affected by any configurable parameter. So, they maintain the same movement path in different simulation instances. Therefore, they are not processed multiple times, and their states in $C(1,1)$ are shared with all other clones.

A cloning tree is generated based on the computation sharing, i.e., child clone in the tree reuses the processing results of parent clone. For example, $C(2,1)$ reuses the processing results of agents $a_3^{(1,1)}$ and $a_4^{(1,1)}$ in $C(1,1)$. A cloning tree for this example is

shown in Figures 5.1(e). Since the processing of child clone depends on its parent clone, the child clone can only be processed after the parent clone finishes its processing.

5.2.3 Top-down cloning strategy

Motivated by the PDES cloning strategy proposed in [50], the top-down cloning strategy generates clones dynamically as the simulation proceeds. The strategy starts the exploration with one simulation instance evaluating the default value choices of parameters (e.g., $C(1,1)$ in Figure 5.1(a)). As the simulation proceeds, when an agent in the default simulation instance first senses alternative value choices of a parameter, cloned simulation instances (i.e., clones) are dynamically generated to evaluate these alternatives. Similarly to the PDES cloning, the cloning decision is made at the same time when an agent is processed, specifically, at sensing stage.

5.2.3.1 Agent execution

To facilitate active and passive cloning, each agent maintains two sets of flags: *cloningFlags* and *clonedFlags*. The cloningFlag indicates to which cloned simulation instance the agent should be cloned and added to; and the clonedFlag keeps track of the cloned simulation instance to which an agent has already been cloned and added. For both sets of flags, there is one flag for each cloned simulation instance for each agent. Taking the parameter exploration task in Figure 5.1 as an example, there are four flags in cloningFlags and clonedFlags, respectively, for each agent.

Two sets of flags are needed to decouple the cloning decision making and actual agent creation, because agent creation can interrupt the lock-stepped agent processing with GPU threads. Thus, the flags are set instead of directly creating agents. In the current time step, agents that will be cloned in the next time step are identified by setting their corresponding cloningFlags. When the agents are actually cloned in the beginning of the next time step, their cloningFlags will be cleared and the corresponding clonedFlags will be set to true accordingly.

For detecting active cloning conditions, an agent checks if a parameter is in its sensing range. Assume the current cloned simulation instance is $C(v_1, v_2, \dots, v_i, \dots, v_{n_p})$. if an

agent senses the i^{th} parameter p_i , all value choices of p_i other than v_i will be evaluated in clones

$$C(v_1, v_2, \dots, v'_i, \dots, v_{n_p}), \text{ where } v'_i \in [1, n_i] \text{ and } v'_i \neq v_i.$$

The agent's cloningFlags corresponding to all $C(v_1, v_2, \dots, v'_i, \dots, v_{n_p})$ will be set to true. This is to indicate that the agent will be cloned in the next time step in order to evaluate other value choices of the parameter. For the example shown in Figure 5.1, when agent $a_1^{(1,1)}$ in C(1,1) senses parameter p_1 , it will set its cloningFlags corresponding to C(2,1) to true. A cloned agent $a_1^{(2,1)}$ will be generated to evaluate the second value of parameter p_1 . Later when agent $a_1^{(2,1)}$ in C(2,1) senses parameter p_2 , it will set its cloningFlags corresponding to C(2,2) to true. A cloned agent $a_1^{(2,2)}$ will be generated to evaluate the second value of parameter p_2 .

For detecting passive cloning conditions, an agent checks all its neighbors' clonedFlags. If a neighbor's clonedFlag corresponding to a clone C' is true, the agent will set its own cloningFlag for the clone C' to true. This is to indicate that the state of this agent will be influenced by a cloned neighbor in the clone C' . So, it needs to create a copy of itself to participate in the execution of the clone C' . For example, as shown in Figure 5.1(b), assume agents $a_2^{(1,1)}$ has been actively cloned to C(2,1), and the clonedFlag of $a_2^{(1,1)}$ corresponding to C(2,1) is true. When $a_3^{(1,1)}$ senses $a_2^{(1,1)}$ in its neighborhood, by reading $a_2^{(1,1)}$'s clonedFlags, $a_3^{(1,1)}$ will set its cloningFlag corresponding to C(2,1) to true.

It is necessary to perform interaction based cloning (i.e., passive cloning) to maintain the correctness of a simulation. However, interaction also makes the passive cloning quickly propagate to agents which have not yet been impacted by the variation of parameter values. A throttling mechanism to refrain cloning propagation is therefore proposed. It invokes an extra round of *compare-and-eliminate* in each time step. Suppose two values of a parameter are evaluated by two clones, C and C' . The clone C is the parent clone, and C' is the child of C. State difference between an agent $a_k^{C'}$ in the clone C' and its counterpart a_k^C in the clone C is evaluated. If the difference between a_k^C and $a_k^{C'}$ lies within a predefined threshold T (which can be defined by modeler), agent $a_k^{C'}$ will be removed from the clone C' . This is because the difference is so small that the state of a_k^C in the parent clone C is sufficient to represents the agent state under the new parameter setting.

5.2.3.2 Clone management

In top-down cloning strategy, cloning decision is made by the parent clone and the cloning tree is constructed dynamically during simulation. When an agent in the parent clone first senses a parameter, a child clone will be generated and the agent will be cloned and added to the child clone to evaluate the alternative value choices of the sensed parameter.

The **AgentPool** and the **Context** introduced in Chapter 3 can support the incremental cloning. The actual agents are kept in the **AgentPool**. The **Context** only contains references to agents and it may contain references to agents in **AgentPools** of different clones.

AgentPool of a child clone is updated by scanning through agents in the **AgentPool** of the parent clone. If an agent in the parent clone's **AgentPool** has true cloningFlags, the agent will be cloned and added to the **AgentPool** of the child clone.

To correctly capture the simulation state of a cloned simulation instance, a **Context** needs to be created and updated. An agent will use the **Context** to perform sensing to update its state accordingly. The clone tree structure facilitates the creation and update of the **Context** as well as computation sharing. Processing on agents that have not been influenced by the parameter variation yet is not required in the child clone. The states of these agents can be directly inherited from parent and ancestor clones. Hence, in such a way, the cloned agents can interact with non-cloned agents in parent and ancestor clones.

In Figures 5.1(a)-5.1(d), the full circles represent agents of the current clone, which are kept in the **AgentPool** (e.g., $a_1^{(2,1)}$ is in the **AgentPool** of C(2,1)). The dotted circles represent agents belonging to the parent or ancestor clones of the current clone. The **Context** of the current clone keeps the references to the agents in the current clone's **AgentPool** (e.g., $a_1^{(2,1)}$ of C(2,1) is in the **Context** of C(2,1)) and also the references to the agents in the parent and ancestor clones that are not yet cloned (e.g., $a_3^{(1,1)}$ of C(1,1) is also in the **Context** of C(2,1)).

The **Context** is created in a top-down manner. A child clone first copies the **Context** of its parent, and then uses the references of agents in its own **AgentPool** to replace the references of the corresponding agents in the **Context**. Using Figure 5.1 as an example, initially there are four agents in C(1,1), $a_1^{(1,1)}$, $a_2^{(1,1)}$, $a_3^{(1,1)}$ and $a_4^{(1,1)}$. Suppose at time t_1 , $a_1^{(1,1)}$ senses p_1 . A simulation clone C(2,1) is created and agent $a_1^{(1,1)}$ is actively cloned

(i.e., agent $a_1^{(2,1)}$ is created in clone $C(2,1)$). At a later time t_2 , $t_2 > t_1$, when agent $a_1^{(1,1)}$ in $C(1,1)$ senses p_2 , a simulation clone $C(1,2)$ will be created and agent $a_1^{(1,1)}$ will be actively cloned again (i.e., agent $a_1^{(1,2)}$ will be created in clone $C(1,2)$). Assuming at the same time, agent $a_2^{(1,1)}$ senses p_1 , it will be actively cloned to $C(2,1)$ (i.e., agent $a_2^{(2,1)}$ will be created in $C(2,1)$). At time t_3 , $t_3 > t_2$, when agent $a_1^{(2,1)}$ in $C(2,1)$ senses p_2 , a simulation clone $C(2,2)$ will be created and agent $a_1^{(2,1)}$ will be actively cloned (i.e., agent $a_1^{(2,2)}$ will be created in clone $C(2,2)$). Both agents $a_3^{(1,1)}$ and $a_4^{(1,1)}$ are neither actively nor passively cloned.

The **AgentPools** and **Contexts** of all clones at t_3 are shown in Table 5.1. The **Context** of $C(1,1)$ is $a_1^{(1,1)}$, $a_2^{(1,1)}$, $a_3^{(1,1)}$ and $a_4^{(1,1)}$. The clone $C(2,1)$ copies the **Context** of $C(1,1)$, and then replaces $a_1^{(1,1)}$ and $a_2^{(1,1)}$ with $a_1^{(2,1)}$ and $a_2^{(2,1)}$ respectively to create its **Context**. Similarly, $C(1,2)$ copies the **Context** of $C(1,1)$, and then replaces $a_2^{(1,1)}$ with $a_2^{(1,2)}$; and $C(2,2)$ copies the **Context** of $C(2,1)$, and then replaces $a_1^{(2,1)}$ with $a_1^{(2,2)}$ to create its **Context**. In this way the computation of agents that have not yet cloned in the parent clone needs not to be repeated in the child clone. Agents $a_1^{(2,1)}$ and $a_2^{(2,1)}$ in clone $C(2,1)$ can access $a_3^{(1,1)}$ and $a_4^{(1,1)}$ of $C(1,1)$ in the **Context** of clone $C(2,1)$; and from the **Context** of clone $C(2,2)$, $a_1^{(2,2)}$ in $C(2,2)$ can access $a_3^{(1,1)}$ and $a_4^{(1,1)}$ of $C(1,1)$, and $a_2^{(2,1)}$ of $C(2,1)$.

 Table 5.1: **AgentPool** and **Context** of scenario in Figure 5.1

Clone	AgentPool	Context
$C(1,1)$	$a_1^{(1,1)}, a_2^{(1,1)}, a_3^{(1,1)}, a_4^{(1,1)}$	$a_1^{(1,1)}, a_2^{(1,1)}, a_3^{(1,1)}, a_4^{(1,1)}$
$C(2,1)$	$a_1^{(2,1)}, a_2^{(2,1)}$	$a_1^{(2,1)}, a_2^{(2,1)}, a_3^{(1,1)}, a_4^{(1,1)}$
$C(2,2)$	$a_1^{(2,2)}$	$a_1^{(2,2)}, a_2^{(2,1)}, a_3^{(1,1)}, a_4^{(1,1)}$
$C(1,2)$	$a_1^{(1,2)}$	$a_1^{(1,2)}, a_2^{(1,1)}, a_3^{(1,1)}, a_4^{(1,1)}$

The **Context** is updated in a similar manner. In the scenario in Figure 5.1, assume at a later time t_4 , agent $a_3^{(1,1)}$ in clone $C(1,1)$ senses $a_2^{(1,1)}$ and is passively cloned. A cloned agent, $a_3^{(2,1)}$, is created in clone $C(2,1)$. The **Contexts** of $C(2,1)$ and $C(2,2)$ will be updated to $\{a_1^{(2,1)}, a_2^{(2,1)}, a_3^{(2,1)}, a_4^{(1,1)}\}$ and $\{a_1^{(2,2)}, a_2^{(2,1)}, a_3^{(2,1)}, a_4^{(1,1)}\}$ respectively. However, this mechanism for clone **Context** creation and update stipulates a specific ordering of clone execution. Child clones can only be processed when its parent clone finishes processing. But, the execution of sibling clones can be potentially concurrent.

Section 6.2 demonstrates that the algorithm produces correct answer when agents sense the parameter in the same sequence. When agents sense parameters in the same sequence, there is a unique way to generate the cloning tree, and the agents to be cloned to the child clone must be in the **AgentPool** of the parent clone. In such circumstances, agents must be correctly cloned.

5.3 GPU implementation

Cloning requires frequently creating newly cloned agents. As discussed in Chapter 3, native dynamic memory allocation on GPU is not efficient. To accommodate our design requirements, the **AgentPool** is adopted to manage the creation of the cloned agents. The details of **AgentPool** can be found in Chapter 3.

5.3.1 Overall workflow

Algorithm 4 is the overall per-step workflow of the top-down cloning strategy for a parameter space exploration task. The clone evaluating the default value choices of parameters (denoted as the **defaultClone**) is initiated with agents in the beginning. In line 2, **defaultClone** performs agent step and cloning condition checking (see lines 14 to 18, which will be further discussed in Algorithm 5) to determine if any agents of the **defaultClone** need to be cloned and added to the child clones of the **defaultClone**. The child clones of the **defaultClone** are added to a clone list (line 3). In top-down approach, a child clone is created dynamically when its first agent is actively cloned. The **childClones** method checks agents in the **AgentPool** of the parent clone in parallel to determine if new child clones need to be created. If not, it just returns the existing child clones.

Then, the cloning tree is traversed in a breath-first manner, level by level. The while-loop (lines 4 to 12) steps through the levels of the cloning tree and the for-loop (lines 5 to 10) processes the clones at the same level of the cloning tree. Suppose the current size of clone list is n_l . The clones in the clone list will be processed one at a time. For each clone in clone list, it adds new cloned agents to itself according to the flags of agents in its parent clone (in line 6, which will be further discussed in Algorithm 6). Then, the clone performs agent processing and cloning condition checking (in line 7) by calling the

Algorithm 4: Overall workflow of the top-down cloning strategy

```
1 Function task.step()
2   defaultClone.step();           // Process default clone
3   cloneList.add(defaultClone.childClones()); // Next batch of clones
4   while cloneList is not empty ( $n_l$  is the current size of clone list) do
5     for  $i = 1$  to  $n_l$ , clone = cloneList[i] do
6       clone.performClone(clone.parent); // Create cloned agents
7       clone.step();                     // Process current clone
8       clone.compareAndElimiate();       // Throttling mechanism
9       cloneList.add(clone.childClones()); // Next batch of clones
10    end
11    cloneList.remove(1,  $n_l$ );
12  end
13 end
14 Function clone.step()
15   foreach agent in clone.AgentPool do in parallel
16     // Process agents and perform cloning condition detection
17     agent.stepAndCheckCloningCondition(clone.Context);
18   end
19 end
```

function `clone.step` (see lines 14 to 18) to determine if any agents in its `AgentPool` need to be cloned. Line 8 invokes the throttling mechanism. In line 9, child clones of the current clone that will be processed in the next batch are returned by `childClones` methods, and they are added to the tail of the clone list. In line 11, clones processed in the current batch (with index from 1 to n_l) are removed from the clone list.

5.3.2 Cloning condition checking

Given the GPU threads, each agent in the `AgentPool` is processed using a thread. Each agent keeps two sets of flags as a part of agent data: `cloningFlags` and `clonedFlags`. With a large number of agents and clones, storing these two sets of flags requires a large memory space. In our implementation, these flags are compressed into bits. Taking `cloningFlags` as an example, in a 32 bit machine, they are stored as an array of 32-bit integers. The size of the array is determined by the number of parameters n_p , one integer per parameter. Thus, each parameter is allowed to have 32 value choices. For the parameter p_i , n_i out of 32 bits will be used to represent n_i value choices of p_i . By default, the first bit is set to

Algorithm 5: Agent's cloning condition checking embedded in agent step function

```

1 Function agent.stepAndCheckCloningCondition(context)
  // Agent senses parameters and performs active cloning
2  foreach parameter  $p_i$  do
3    if agent.canSense( $p_i$ ) then
4      agent.performThink( $p_i$ );
      // Set first  $n_i$  bits of integer corresponding to  $p_i$  to 1
5      setBits(agent.cloningFlags[i],  $n_i$ );
6    end
7  end
  // Agent interacts with neighbors and perform passive cloning
8  rect = context.initNeighbors(agent.location, agent.sensingRadius);
9  neighbor = context.nextNeighbor(rect);
10 while neighbor  $\neq$  NULL do
11   agent.performThink(neighbor);
   // Update agent's CloningFlags with neighbor's clonedFlags
12   foreach parameter  $p_i$  do
13     agent.cloningFlags[i] = agent.cloningFlags[i]  $\vee$  neighbor.clonedFlags[i];
14   end
15   neighbor = context.nextNeighbor(rect);
16 end
17 agent.performAct();
18 end

```

‘1’ and the rest are set to ‘0’. When an agent will be either actively or passively cloned because of the v_i -th value choice of the i^{th} parameter p_i , the v_i -th bit of cloningFlags[i] will be set to ‘1’.

For active cloning condition checking, an agent checks if a parameter is in its sensing range. If the agent senses the parameter p_i , the first n_i bits of cloningFlags[i] are set to ‘1’, indicating that multiple copies of the agent will be created to evaluate n_i value choice of the parameter p_i . For passive cloning condition checking, each agent performs a bitwise OR operation between its cloningFlags and neighbors’ clonedFlags. This will set the corresponding bits in an agent’s cloningFlags to ‘1’ if it encounters cloned neighbors.

An agent’s Sense-Think-Act cycle is typically implemented in the **agent.step** function that is executed in every time step. To incorporate the cloning mechanisms described above, the **step** function is modified, as shown in Algorithm 5. It is performed concurrently by each agent in a clone’s **AgentPool**. The active cloning condition checking is

Algorithm 6: Actual creation of cloned agents

```
1 Function clone.performClone(parentClone)
   // Copy Context from parent and update it
2   copy(clone.Context, parentClone.Context);
3   updateContext(clone.Context, clone.AgentPool);
4   foreach parentAgent in parentClone.AgentPool do in parallel
       // Suppose current clone and its parent differ at  $p_i$ .
       // Create agent if it is influenced but has not been cloned.
5       if clone.cloneMask[i]  $\wedge$  parentAgent.cloningFlags[i]
            $\wedge$   $\neg$ parentAgent.clonedFlags[i] then
6           parentAgent.clonedFlags[i] = parentAgent.clonedFlags[i]
                $\vee$  clone.cloneMask[i];
7           childAgent = cloneFrom(parentAgent);
8           childAgent.origin = parentAgent;
9           clone.AgentPool.add(childAgent);
10      end
11  end
12 end
```

to determine if a configurable parameter is in the sensing range of the agent (lines 2 to 7). Agent also performs thinking with the sensed parameter (line 4). The neighbor interaction and passive cloning condition checking are performed from lines 8 to 16. In line 13, an agent sets its cloningFlags to indicate that it will be cloned when it detects a cloned neighbor. The agent updates its cloningFlags by performing a logic OR operation with neighbor's clonedFlags. Finally, the agent updates its state in line 17 by calling the function `performAct`.

As explained in Section 5.2.3.1, setting cloningFlags only indicates that agents will be cloned in the next time-step. Copies of agents whose cloningFlags are set will be actually created at the beginning of the next time-step.

5.3.3 Actual agent creation

Each clone maintains a cloneMasks array with n_p integers as a part of clone properties. It represents parameter value choices evaluated by the clone. For example, given a clone $C(a,b)$ evaluating two parameters, where $1 \leq a \leq n_1$ and $1 \leq b \leq n_2$, its cloneMasks array has the a^{th} bit of the first integer and the b^{th} bit of the second integer set to '1'.

Assume the parameter value combinations of parent and child clones differ at parameter p_i , $1 \leq i \leq n_p$. To determine whether or not a copy of an agent in the parent clone needs to be created in the child clone, a bitwise **AND** operation needs to be performed between the agent's `cloningFlags[i]` and the child clone's `cloneMasks[i]`. For example, given a clone $C(2,1)$ and its parent clone $C(1,1)$, whether or not an agent in $C(1,1)$ needs to be cloned and added to $C(2,1)$ can be checked by performing a bitwise **AND** operation between `cloningFlags[1]` of the agent and `cloneMasks[1]` of the clone $C(2,1)$. If the result is not 0, it indicates that the second bit of `cloningFlags[1]` must be '1'. So, the agent has already been impacted by the second value choice of p_1 , and a copy of this agent needs to be created in the $C(2,1)$.

The pseudo-code of this procedure is given in Algorithm 6. First, child clone copies the **Context** of parent clone and update the **Context** with agents in its **AgentPool** (lines 2 and 3). The actual agent creation procedure is shown in lines 4 to 11. Each agent in the **AgentPool** of parent clone will be checked (line 4). Assume the parameter value combinations evaluated by parent clone and child clone differ at parameter p_i . Line 5 specifies the agent creation condition, that is, agent in the parent clone will be cloned if the agent is influenced by current clone's value choices and it has not been added to current clone. If satisfied, the agent in the parent clone updates its i^{th} `clonedFlag`, indicating it has been cloned (line 6), and the new agent is created and added to the child clone (lines 7 to 9).

5.3.4 Throttling mechanism

To refrain passive cloning propagation, a throttling mechanism is implemented as shown in Algorithm 7. If the difference between an agent in the child clone and its origin in the parent clone is within threshold T , the former will be removed and the latter's corresponding bit in `clonedFlags[i]` is cleared (i.e., changed to 0). Since the `clonedFlags[i]` may be modified by threads corresponding to agents in multiple child clones concurrently at different bits, the bitwise **AND** operation needs to be performed atomically.

Algorithm 7: Refraining passive cloning propagation

```

1 Function clone.compareAndEliminate()
2   foreach agent in clone.AgentPool do
3     diff = compareDiff(agent, agent.origin);
4     if diff < T then
5       clone.AgentPool.remove(agent);
6       // Suppose current clone and its parent differ at  $p_i$ .
7       atomicAnd(agent.origin.clonedFlags[i],  $\neg$ clone.cloneMask[i]);
8     end
9   end

```

Figure 5.2: Groups of concurrent clones and synchronization barriers

5.3.5 Optimization

5.3.5.1 Concurrent clone execution

In our implementation, each agent is processed using a thread. The agent's step function is triggered by `clone.step` function which is implemented as a GPU kernel function (see lines 14 to 18 in Algorithm 4). To further increase concurrency, the `clone.step` function can be invoked on multiple cloned simulation instances concurrently with multiple CUDA streams. Executions of GPU kernel functions scheduled on different CUDA streams can be overlapped. Therefore, not only agents of the same clone are processed concurrently, multiple clones are also processed concurrently.

Clones at the same level in the clone tree are either child clones of the same parent clone or of different parent clones. They are independent and can be processed in parallel (i.e., lines 5 to 10 in Algorithm 4 can be processed in parallel). However, processing of a child clone depends on the execution results of the parent clone (as

explained in Section 5.2.3.2). So, clones are grouped by levels in the clone tree. A synchronization barrier need to be placed between upper and lower levels to make the child clones wait for their parent clones finishing processing (for example shown in Figure 5.2). The barrier is inserted between line 10 and line 11 in Algorithm 4. A CUDA API, `cudaStreamSynchronize`, can be used to implement the synchronization barrier.

5.3.5.2 Sequential operation aggregation

As explained above, removing holes in an `AgentPool` and preprocessing `Context` for efficient neighbor searching require sorting operations. Although `Thrust` library¹ is adopted to carry out highly optimized GPU based sorting algorithm, this part of processing is still very time consuming and accounts for a big percentage of the total execution time. Besides, each sorting operation occupies the entire GPU, which prevents the use of streams to achieve concurrent processing. Perform sorting sequentially for each clone under-utilizes GPU.

To tackle such inefficiency, a mechanism is proposed to aggregate data sets of all the clones and then sorting is performed on the aggregated data set. Each clone copies the data into its reserved segment of a bigger array. Then each data element is labeled with the clone id. During actual sorting, the clone id is also included as a key. Consequently, data of the same clone are still grouped together after sorting and thus can be conveniently copied back to the correct clone for processing later.

5.4 Experiments

A case study to demonstrate the performance gain obtained through ABS cloning was conducted. The test case is a pedestrian evacuation simulation implemented using social force model [42]. In this case study, agents are initiated in room 1 and will be evacuated to room 4 through three gates, as shown in Figure 5.3. The size of each gate is a configurable parameter to be evaluated with the cloning mechanism. The details of the experiment setups are given in Table 5.2. The experiment was performed on a 1.25GHz Fermi-based NVIDIA Quadro 2000 GPU, whose specifications and setups are described in Section 3.4.

¹<https://developer.nvidia.com/Thrust>

Table 5.2: Experiment setups

Description	Value
Environment dimension (unit)	128 * 128
Gate 1, 2, and 3 size choices (unit)	2, 3, 4, 5, 6, 7
Max agent speed (unit per step)	3
Number of time steps	1000
size of a time step	0.1 second
Threshold value in throttling mechanism	DBL_EPSILON (2.22e-16)

Figure 5.3: Scenario of the case study Figure 5.4: Correctness verification

5.4.1 Verification

Simulation cloning reduces computation but should guarantee the correctness. The correctness is defined as that the cloned execution should generate the same result as the stand-alone execution. To show the correctness, the results of clones in ABS cloning and their stand-alone counterparts were compared both with and without the throttling mechanism. Since there are three parameters (i.e., gates), each with three different value choices, the total number of clones is 27. Only the result of the 27th clone, which takes the last value choice of each parameter, is used to compare against the execution result of its stand-alone counterpart. This clone is chosen because it has fewest agents of its own and thus its correctness highly depends on all its ancestor clones. To verify the correctness of our cloning mechanism, the number of agents passing through the last gate at each simulation time-step is used as a metric. Randomness is removed to guarantee multiple simulation runs of the same parameter configuration have identical simulation result.

Table 5.3: Profiled execution time of various optimization cases at a time step

Optimizations (with throttling mechanism enabled)	Execution time of a time step	
	Default Clone	All Clones
Opt 1 off, Opt 2 off	4.145 ms	120.235 ms
Opt 1 on, Opt 2 off	4.352 ms	100.180 ms
Opt 1 on, Opt 2 on	4.346 ms	49.259 ms

The test case with 1024 agents running 1000 steps shows that the per step difference between the two execution results are 0, as shown in Figure 5.4. The state of each agent (location, velocity, etc.) in every simulation time-step is also the same between the two executions. In particular, the cloned simulation using throttling mechanism with threshold value of `DBL_EPSILON` (i.e., `T_on` in Figure 5.4) also generates the same result. This small threshold makes the correctness of simulation remain intact. As shown by lines 1 and 2 in Figure 5.5, the throttling mechanism significantly improves execution performance.

5.4.2 Benefit of optimization

This part of experiments shows the performance enhancement obtained through the two optimization techniques, i.e., concurrent clone execution (denoted as *opt 1*) and sequential processing aggregation (denoted as *opt 2*). Same as Section 5.4.1, the test case with 1024 agents running 1000 steps was used in the experiments. Throttling mechanism was also used in the experiments. Table 5.3 shows the profiled execution time of the default clone (i.e., the simulation instance that takes the first value choice of each parameter) and the total execution time of all clones at the 800th time step under various optimization cases. NVIDIA NSight Profiler v4.5 was used to collect the profiled execution time.

In all three cases, the step function of the default clone takes almost the same time to finish (~ 4 ms). When both optimization techniques are not used, all clones are processed sequentially and individually. The total execution time of the step is ~ 120 ms. When concurrent clone execution is turned on, all 27 clones are processed using different CUDA streams. The synchronization barriers are set between the child and parent clones and processing of clones at the same level of the clone tree are overlapped. As a result, the

Figure 5.5: Performance of optimization

total execution time of the step is reduced to ~ 100 ms (i.e., 1.2 times speed up). When both optimization techniques are used, the total execution time of the step is further reduced to ~ 49 ms, resulting 2 times speed up.

Figure 5.5 shows the per-step execution time of four cases obtained with `cudaEventRecord`, i.e., the three cases shown in Table 5.3 plus the case without using the throttling mechanism and optimization techniques. Jumps in curves correspond to the moments when agents sense a new parameter. The computation load in each simulation time-step is determined by two factors: i) the total number of agents in the `AgentPools` of all the cloned simulation instances; and ii) the amount of processing per agent. Without using the throttling mechanism, agents will be cloned rapidly to all the cloned simulation instances and the total number of agents will reach the maximum before simulation time-step 400. As shown in Figure 5.3, in the test case used in the experiments, the execution time per step of each clone will decrease as simulation progresses since the density of the crowd surrounding gate 1 decreases. This results in the decreasing trend of per step execution time, after it reaches the maximum, for the case where the throttling mechanism is not used. In all other cases, the throttling mechanism is on and the total number of cloned agents gradually increases and never reaches the maximum.

The results shown in Figure 5.5 (obtained with `cudaEventRecord`) are consistent with that in Table 5.3 (obtained with NVIDIA NSight Profiler). Particularly, at the 800th time step of Figure 5.5, the execution time of the case with both opt 1 and opt 2 disabled is

slightly longer than that with opt 1 enabled and opt 2 disabled, and the execution time of these two cases almost doubles that with both opt 1 and opt 2 enabled. In general, for the case where both optimization techniques are used, it does not perform well at the beginning when most of the `AgentPools` are empty. However, it significantly outperforms all the other cases as simulation progresses.

5.4.3 Overall performance gain

In this part of experiments, the overall performance of the proposed ABS cloning technique is evaluated. All optimizations are turned on. The execution time of stand-alone simulations is used for comparison. Speedup is computed as follows:

$$speedup = \frac{Time_{standalone} \times Num_{clones}}{Time_{sim_cloning}}$$

The total execution time of all stand-alone simulations is estimated by multiplying the execution time of one stand-alone simulation by the number of clones.

First, experiments were conducted to show the performance of the cloning strategy with respect to the number of agents in the simulations. The number of agents in the default simulation instance were varied from 128 to 2048. 27 parameter value combinations were evaluated with simulations using the proposed cloning technique. The sizes of gates evaluated with the 27 clones have three choices: 2, 3, and 4. For comparison, two stand-alone simulations were executed: the size of all three gates was set to 2 in the first stand-alone simulation (denoted as SA_2), and the sizes of all three gates were set to 7 in the second stand-alone simulation (denoted as SA_7).

Figures 5.6(a) and 5.6(b) show the overall execution time and speedups of 1000 steps. Figure 5.6(a) shows that large gate sizes can slightly reduce the execution time (the execution time of SA_7 is slightly shorter than that of SA_2). Figure 5.6(b) shows that the proposed simulation cloning technique achieves speedups in comparison with using stand-alone simulations to evaluate each of 27 parameter value combinations.

Second, to show the performance of the cloning technique with respect to the number of clones, the number of agents was fixed to be 512 and the number of value choices of each of the three gates were 2 (gate sizes: 2 and 3), 3 (gate sizes: 2, 3, 4) and 4 (gate sizes: 2, 3, 4 and 5), resulting 8 (i.e., 2^3), 27 (i.e., 3^3) and 64 (i.e., 4^3) clones. The cloning

Figure 5.6: Execution time and speedup vs. number of agents

technique was compared with the stand-alone simulation with gate size to be 7 (denoted as SA_7).

Figures 5.7(a) and 5.7(b) show the per-step execution time and speedups of multiple clones running for 1000 steps. When the number of clones increases, the execution time increases accordingly. The maximum per-step speedup is obtained at the beginning stage of the simulation, because agents are not yet cloned. The computation performed by the parent clone is shared by all child clones. However, this computation sharing will decrease when the number of cloned agents increases, resulting in the decrease of per-step speedup. The total execution time and average speedups are shown in Figure 5.7(c) and Figure 5.7(d). The best average speedup of 2.53 is achieved when cloning technique is used to evaluate 64 parameter configurations.

5.5 Summary

To summarize, in this chapter, a top-down cloning strategy is proposed for ABS and its implementation on GPU was presented. To support incremental cloning of ABS, mechanisms for both actively and passively cloning agents are proposed. A scheme to maintain the correct **Context** for each cloned ABS instance is developed. In addition, a strategy to restrain the propagation of passive cloning in order to maximize computation sharing amongst cloned ABS instances is also investigated. In addition, the implementation is further optimized using concurrent clone execution and sequential operation aggregation

Figure 5.7: Execution time and speedup vs. number of clones

techniques. Experiments were conducted using an evacuation simulation to demonstrate the correctness and performance advantages of our approach. The cloning approach achieves significant speedup, compared to the stand-alone executions, when evaluating total multiple parameter configurations.

The top-down cloning strategy has merits but it also has limitations that the child clone is different from the parent clone on only one parameter, and agents need to sense parameter in the same sequence. In the next chapter, the cause and the impact of these limitations are further discussed. To overcome these limitations, a bottom-up cloning strategy is proposed, which maintains the efficiency and can also support more ABS scenarios.

Chapter 6

A Bottom-up ABS Cloning Strategy

6.1 Overview

The top-down cloning strategy starts the exploration with one simulation instance evaluating the default value choices of parameters. Whenever an agent senses a different parameter configuration, simulation instances evaluating other value choices of the parameter are cloned from the default instance. Multiple copies of the agent are created and added to the cloned simulation instances. The processing results of non-cloned agents in the default simulation instance can be shared with the cloned simulation instances. A cloned simulation instance can be further cloned when the agent in the cloned simulation instance encounters other configurable parameters.

To maximize the computation sharing, the proposed top-down cloning heuristic enforces that a child clone can have only one parameter value different from its parent. If the parameter value combination evaluated by a simulation instance C has two or more values different from the combination evaluated by the default simulation instance, several intermediate clones need to be generated before C . This and other limitations of the top-down cloning strategy are further elaborated in Section 6.2.

Generating and processing intermediate clones incur extra overheads and it contradicts the goal of using cloning to reduce the execution time. In this chapter, a bottom-up cloning strategy is proposed to overcome the limitations of the top-down cloning strategy. In the bottom-up cloning strategy, cloned simulation instances evaluating the given combinations of parameter values are created before execution. In contrast to the top-down cloning strategy, the cloning tree organizing the cloned instances is also predetermined.

Simulation instance at root is regarded as the default instance and contains all agents at the beginning. Agents are cloned and added to the cloned simulation instances based on the predetermined cloning tree.

Given several simulation instances evaluating different parameter value combinations, one simulation instance can share some computations with other simulation instances. The amount of computation sharing in ABS is measured by the number of agents that have the identical state in the two simulation instances. A complete graph can be formed with all possible sharing relationships. The vertices are simulation instances. The edges indicate possible parent-child clone relationship between simulation instances. Each edge has a weight representing the estimation of the amount of computation sharing between the two simulation instances. According to Cayley's formula [12], total $n_c^{n_c-2}$ spanning trees can be constructed from a complete undirected graph, where n_c is the number of clones. Among all these spanning trees, there exists one spanning tree that maximizes computation sharing among all clones. To determine the optimal cloning tree, each of the $n_c^{n_c-2}$ cloning trees needs to be executed. Obviously, this is not a viable solution. In this chapter, a minimum spanning tree (MST) heuristic is proposed to generate a cloning tree for the bottom-up cloning strategy.

The rest of the chapter is structured as follows: Section 6.2 explains the limitations of the top-down cloning strategy. Section 6.3 describes the detailed design of the bottom-up cloning strategy. Section 6.4 explains the GPU implementation of the bottom-up cloning strategy. Section 6.5 demonstrates the correctness and effectiveness of the bottom-up cloning strategy with a set of case studies of different simulation scenarios. Finally, Section 6.6 summarizes this chapter.

6.2 Issues of the top-down cloning strategy

First, in the top-down cloning strategy, the parameter value combinations evaluated by the parent and child clones differ only at one parameter. For example in Figure 6.1, C(1,1) and C(2,1) differ at p_1 , C(1,1) and C(1,2) differ at p_2 , and C(2,1) and C(2,2) also differ at p_2 . If only (1,1) and (2,2) are parameter value combinations of interests, not only clones C(1,1) and C(2,2) need to be created, but also C(2,1) or C(1,2) needs to be generated as an intermediate clone. Processing intermediate clones incurs overheads.

Figure 6.1: A scenario that top-down cloning strategy is not effective

Second, clones may not be generated correctly if agents sense parameters not in the same sequence. For example in Figure 6.1, agent $a_1^{(1,1)}$ is created at the other side of the environment and moves in the opposite direction of other agents (Figure 6.1(a)). In this scenario, $C(2,1)$ and $C(1,2)$ are generated when agents $a_2^{(1,1)}$ and $a_1^{(1,1)}$ sense p_1 and p_2 respectively (Figures 6.1(b) and 6.1(c)). As simulation progresses, $C(2,2)$ can be generated either when $a_2^{(2,1)}$ senses p_2 (Figure 6.1(d)), or when $a_1^{(1,2)}$ senses p_1 (Figure 6.1(e)). Which version of $C(2,2)$ will be created depends on whether $a_1^{(1,2)}$ senses p_1 first or $a_2^{(2,1)}$ senses p_2 first. If $a_2^{(2,1)}$ senses p_2 first, the cloning tree will be the one in Figure 6.1(f); otherwise, the cloning will be the one in Figure 6.1(g). However, neither of the two versions of $C(2,2)$ is correct. If $C(2,2)$ is cloned from $C(2,1)$, $a_1^{(1,1)}$ will be in $C(2,2)$'s **Context**. But, $a_1^{(1,1)}$ was updated using the first value choice of p_2 . Similarly, cloning $C(2,2)$ from $C(1,2)$ will not have a correct copy of a_2 in $C(2,2)$'s **Context**. Nonetheless, if all agents sense parameters in the same sequence, this problem will not exist because the clones will be generated in a unique sequence (e.g., the scenario depicted in Figure 5.1). Therefore, agents will be cloned and **Context** will be created correctly.

To address the above-mentioned issues of the top-down cloning strategy, an improved cloning strategy, the bottom-up cloning strategy is proposed accordingly.

6.3 Bottom-up cloning strategy

The bottom-up cloning strategy is different from the top-down cloning strategy in three aspects. First, in the top-down cloning strategy, cloning condition checking is performed by agents in the parent clone. The agent step function in the top-down cloning strategy needs to be modified to integrate the cloning condition checking. An agent in the parent clone checks the environment and surrounding neighbors to determine whether or not it needs to be cloned simultaneously when it performs the Sense-Think-Act cycle (as discussed in Algorithm 5 in Section 5.3.2). In contrast, in the bottom-up cloning strategy, the child clone performs the cloning condition checking to determine if an agent in its parent or ancestor clone needs to be cloned and added to itself. Therefore, agent processing in the parent clone and the cloning condition checking in the child clone are separated. This design of the bottom-up cloning strategy leads to a benefit that the agent step function does not need to be modified to integrate the cloning related logics.

Second, in the top-down cloning strategy, only agents in the **AgentPool** of the parent clone will be considered to be cloned. This is because the cloning condition checking is coupled with agent processing, as described above, and only agents in the **AgentPool** are processed. In the bottom-up cloning strategy, cloning decisions are performed by checking agents in the **Context** of the child clone. The **Context** contains agents not only belonging to the current clone, but also agents belonging to the parent and ancestor clones. Therefore, the child clone can determine which agent in parent and ancestor clones needs to be cloned and added to its **AgentPool**. It should be noted that cloning condition checking in bottom-up cloning strategy incurs more overhead than that in the top-down cloning strategy, because the number of agents in the **Context** is more than that in the **AgentPool**. This overhead is considered and minimized in our implementation of the bottom up cloning strategy, which will be discussed in Section 6.4.

Third, instead of constructing the cloning tree dynamically and simultaneously with agent processing, the bottom-up cloning strategy generates all the clones and determines the cloning tree before execution. The construction of the cloning tree is discussed in the following subsection.

6.3.1 Constructing cloning tree - an MST heuristic

Since the major workload in ABS is agent processing, the computation sharing of ABS between two simulation instances is measured by the number of the shared agents. The number of shared agents changes over time. As the simulation goes on, more and more agents will exhibit different behaviors in different clones due to the influences of various parameter value combinations. Consequently, fewer and fewer agents can be shared.

In the bottom-up cloning, computation sharing drives the formation of parent-child clone relationship. Simulation instances evaluating various parameter combinations form a complete graph first, in which vertices are simulation instances evaluating parameter value combinations and edges represent potential parent-child clone relationship. Each edge carries a weight. The weight represents the estimation of the amount of computation sharing between two instances.

According to Cayley's formula [12], given n_c vertice (simulation instances), totally $n_c^{n_c-2}$ spanning trees can be generated. Among all these trees, An optimal tree exists, which maximizes the total computation sharing between all parent-child pairs. The optimal tree is the desired cloning tree.

In this work, an MST heuristic is proposed to generate a cloning tree with low overhead. Given the complete indirect graph as mentioned above, the root of the MST is randomly selected. The edge weight between two simulation instances, $C(v_1, \dots, v_i, \dots, v_{n_p})$ and $C(u_1, \dots, u_i, \dots, u_{n_p})$, where v_i and u_i are two value choices of the parameter p_i in two simulation instances respectively, is determined by the number of parameters that have the same value, which can be computed as:

$$edgeWeight = \sum_{i=1}^{n_p} w_i \times \psi(v_i, u_i), \text{ where } \psi(a, b) = \begin{cases} 1, & \text{if } a = b; \\ 0, & \text{otherwise.} \end{cases}$$

Coefficient w_i is adjustable for parameter p_i according to, for example, the number of agents influenced by parameter p_i . The main idea of the MST heuristic is that simulation instances with similar parameter value choices may have chance of sharing more computations because the scenarios they evaluate are similar. Therefore, they should be connected in the formulated cloning tree.

After the cloning tree is constructed, the **AgentPool(s)** of the root clone will be initialized with all the agents in the environment. Simulation instances with empty

Figure 6.2: The same scenario of Figure 6.1 handled by the bottom-up cloning strategy

AgentPool will be created for other clones in the cloning tree. The agents are gradually cloned from root to other clones as to be described in the next subsection.

6.3.2 Active cloning and passive cloning

Once the cloning tree is constructed, the parent-child clone relationship is also determined. The active cloning in bottom-up cloning strategy is to determine if an agent, which is in the **Context** of the child clone but belongs to the parent or ancestor clone, needs to be added to the child clone. When an agent in the **Context** of the child clone senses a parameter having different value choices in the parent and child clones, a copy of the agent will be created and added to the **AgentPool** of the child clone. For example in Figure 6.2, the clone $C(2,2)$ is the child clone of $C(2,1)$ in the cloning tree generated by the MST heuristic. The values of p_2 in $C(2,1)$ and $C(2,2)$ are different. Assume at time t_1 , agent $a_1^{(1,1)}$ in the **Context** of $C(2,2)$ senses p_2 . As a result, a cloned version of a_1 , i.e., $a_1^{(2,2)}$, is generated and added to $C(2,2)$.

An non-cloned agent in the **Context** of the child clone will be passively cloned if it senses an agent belonging to the child clone. For the above example, when $a_2^{(2,1)}$ in $C(2,2)$'s **Context** senses $a_1^{(2,2)}$ at t_2 , it will be passively cloned, as shown in Figure 6.2(e). A cloned agent $a_2^{(2,2)}$ will be created and added to $C(2,2)$'s **AgentPool**.

By comparing Figure 6.1(d) (i.e., the top-down cloning strategy) and Figure 6.2(e) (i.e., the bottom-up cloning strategy), it clearly shows that the bottom-up cloning strategy can correctly clone agents (i.e., both $a_1^{(2,2)}$ and $a_2^{(2,2)}$ are in $C(2,2)$'s **AgentPool**). However, the top-down cloning strategy does not work correctly in this scenario. The cloning tree with correct state at t_2 is depicted in Figure 6.2(f).

Since agent cloning is carried out by the child clone using its **Context** in the bottom-up strategy, different from the top-down strategy, any two simulation instances that evaluate different parameter value combinations can form the parent-child clone relationship. In addition, a unique cloning tree is generated before the parameter exploration task begins in the bottom-up strategy. So, the second problem of the top-down strategy does not exist either.

6.4 GPU implementation

Similar to the top-down cloning strategy, the bottom-up cloning strategy also exposes multiple levels of parallelism, i.e., inter-clone parallelism and intra-clone parallelism. To be specific, the inter-clone parallelism allows independent clones such as sibling clones in the clone tree to be processed concurrently. The intra-clone parallelism allows agents in one clone to be processed concurrently. GPU is a suitable platform to exploit these two levels of parallelism. To support inter-clone parallelism, clones are concurrently processed with CUDA streams [74]. The synchronization barrier between the parent clone and child clone is realized using `cudaStreamSynchronize`.

In the top-down cloning strategy, agents in the parent clone check the active and passive cloning conditions and dynamically create new cloned simulation instances and clone agents. In the actual implementation on GPU, the cloning condition checking has to be performed with an agent Sense-Think-Act cycle, but the actual cloned agent creation is decoupled from agent cloning condition checking due to the significant overheads of dynamic memory allocation on GPU [106]. Because of the lockstep execution of SIMT, if a single thread performs a costly dynamic memory allocation, all other threads of the same warp will have to be blocked and wait for the thread to finish. Hence, the actual creation of cloned agents is performed after agents finish processing (i.e., at the beginning of the next time-step).

Algorithm 8: Overall workflow of the bottom-up cloning strategy

```
1 Function task.step()
2   rootClone.step();                                // Process root
3   cloneList.add(rootClone.ChildClones());          // Add next batch of clones
4   while cloneList is not empty, ( $n_l$  is the current size of cloneList) do
5     foreach  $i = 1$  to  $n_l$ , clone = cloneList[i] do in parallel
6       // Cloning condition checking and cloned agent creation
7       clone.performClone(clone.parent);
8       clone.step();                                // Process current clone
9       clone.compareAndElimiate();                  // Throttling mechanism
10      // Add next batch of clones
11      cloneList.mutualExclusiveAdd(clone.childClones());
12    end
13    synchronizationBarrier(); // Wait for the current batch to finish
14    cloneList.remove(1,  $n_l$ ); // Remove the first  $n_l$  clones in the list
15  end
16 Function clone.step()
17   foreach agent in clone.AgentPool do in parallel
18     // Process agent only, no cloning related operations
19     agent.step();
20   end
21 end
```

In constrast to the top-down cloning strategy, in the implementation of the bottom-up cloning strategy, the cloning condition checking and actual agent creation are performed together and are separated from agent Sense-Think-Act cycle by design (as described in the beginning of Section 6.3). The implementation details of the bottom-up cloning strategy are described in the following subsections.

6.4.1 Overall workflow

The overall workflow is shown in Algorithm 8. Inter-clone parallelism is also shown in Algorithm 8: In line 5, clones of one batch are processed in parallel; In line 9, clones for the next batch are added to the cloneList in a mutual exclusive way; In line 11, synchronization barrier is set to separate the processing of two batches.

The overall workflow of the bottom-up cloning strategy is similar to that of the top-

Algorithm 9: Cloning condition checking and actual cloned agent creation

```
1 Function clone.performClone(parentClone)
   // Step 1: Copy Context of the parent clone to the current clone
2   copy(clone.Context, parentClone.Context);
3   updateContext(clone.Context, clone.AgentPool);
   // Step 2: Prepare the cloneGrid
4   foreach agent in clone.AgentPool do in parallel
5       cellId = computeCellIdInCloneGrid(agent.position);
6       clone.ClonedGrid[cellId] = true;
7   end
   // Step 3: Check cloning condition for each agent in the Context
8   foreach parentAgent in parentClone.Context and
       not in clone.AgentPool do in parallel
9       if cloningCondition(parentAgent, parentClone, clone) then
10          childAgent = cloneFrom(parentAgent);
11          childAgent.origin = parentAgent;
12          clone.AgentPool.add(childAgent);
13       end
14   end
15 end
```

down cloning strategy as described in Algorithm 4. But, as described above, there is one important difference that differentiates the bottom-up cloning strategy from the top-down cloning strategy: In the bottom-up cloning strategy, the cloning condition checking is coupled with the actual cloning of the agent in the `performClone` function performed by the child clone (in line 6 of Algorithm 8). After the checking and agent cloning, all agents in the clone will perform their step function (in line 17 of Algorithm 8). In comparison, in the top-down cloning strategy, the cloning condition checking is coupled with the processing of agents in the parent clone (in line 16 of Algorithm 4), and is separated from the actual cloning of agents in the child clone (in line 6 of Algorithm 4).

6.4.2 Cloning decision making and actual agent creation

The bottom-up cloning strategy decouples the cloning decision making and agent processing. Thus, an extra round of neighbor interaction, in addition to the one required in the sensing stage of agent processing, is required for passive cloning condition checking.

Algorithm 10: Cloning condition checking with bottom-up cloning strategy

```
1 Function cloningCondition(parentAgent, parentClone, childClone)
  // Step 1: Active cloning condition checking (Check if
  // parentAgent can sense the parameter difference).  $v_i$  and  $u_i$ 
  // are value choices of  $p_i$  in the parentClone and childClone
  // respectively
2  foreach parameter  $p_i$  do
3    if  $v_i \neq u_i \wedge \text{parentAgent.canSense}(p_i)$  then
4      | return true;
5    end
6  end
  // Step 2: Passive cloning condition checking (Check if
  // parentAgent's sensing range intersects with cell marked
  // true).
7  foreach cell in cloneGrid do
8    if cell.isTrue()  $\wedge$  parentAgent.intersected(cell) then
9      | return true;
10   end
11 end
12 return false;
13 end
```

Neighbor searching incurs significant overheads (as discussed in Chapter 3). This additional overhead may affect the goal of using cloning to reduce the execution time. To address this issue, a **ClonedGrid** data structure is proposed to perform passive cloning efficiently. The environment is decomposed into a grid of cells. Each cell records a binary value: 1 (or true) means there are cloned agents in this cell, and 0 (or false) means otherwise. So, by using **ClonedGrid**, passive cloning can be performed by checking if an agent's interaction range intersects any cell that is marked true. Performing passive cloning in such a coarse-grained manner may cause more agents than necessary to be cloned. But this is not an issue because throttling mechanism will later remove agents that need not to be cloned.

The cloning operations are included in **performClone** function as shown in Algorithm 9. It consists of three steps: First, the **Context** of the parent clone is copied to the child clone. The child clone's existing agents in **AgentPool** are used to update the copied **Context** (lines 2 and 3). Second, the **ClonedGrid** is prepared (lines 4 to 7). For each

agent, its cell id is computed based on its position (line 5) and the corresponding cell is marked to be true (line 6). Third, for each agent in the **Context** and no in the **AgentPool** of child clone, its cloning condition is checked (line 9). Then, cloned agents are created and added to the **AgentPool** of the current clone if needed (lines 10 to 12).

The agent's cloning condition checking can be further divided into two sub-steps as summarized in Algorithm 10. First, the active cloning condition checking is performed (lines 2 to 6). The active cloning conditions are: 1) the value choices of a parameter are different in the parent clone and the child clone; and 2) the agent can sense the parameter. The child clone (denoted as C'') only needs to check the parameters with its parent clone (denoted as C'), and does not need to check with its ancestor clones (denoted as C). If an agent a_i^C is in the **Context** of C'' , it means either a_i^C has not yet sensed any parameter, or the parameters it has sensed in C and C' all have the same values. Otherwise, a cloned agent $a_i^{C'}$ would be created and added to C' , and $a_i^{C'}$ instead of a_i^C would be found in the **Context** of C'' because of the inheritance of the **Context**.

Second, passive cloning condition checking is performed (lines 7 to 11). The passive cloning checks if an agent's sensing range intersects with cells marked true in **ClonedGrid**. Finally, if line 12 is reached, which means the agent is neither impacted by a parameter value choice nor interacts with cloned agents, the function returns false, indicating that the agent should not be cloned.

6.5 Experiments

Case studies were conducted to demonstrate the correctness and effectiveness of the proposed cloning strategies. The experiments were conducted on two platforms. The first one is the CPU platform that consists of an 6-core Intel Xeon E5-1650 processor @ 3.5 GHz with a 12 MB cache. The memory size of the CPU platform is 16 GB. The second one is the GPU platform, i.e., a Fermi-based NVIDIA Quadro 2000 GPU whose specifications and setups are described in Section 3.4.

The simulation applications evaluate agent movement in continuous space with various environmental configurations. The configurable environmental settings are gates of changing sizes and existences of obstacles. The agent interactions are simulated with social force model [42].

Table 6.1: Experiment setups

Scenarios	Dimension (unit)	Moving speed (unit/step)	Time steps	Parameter to evaluate
Scenario 1	128 * 128	3	1000	Gate sizes in unit (three choices)
Scenario 2	64 * 64	3	300	Existence of obstacle (two choices)

6.5.1 Verification

Two scenarios are used to verify the correctness of the top-down cloning strategy and the bottom-up cloning strategy, as shown in Figure 6.3. Both scenarios have 32 agents in the default simulation instance. The details of experiment setups is given in Table 6.1. In the first scenario (Figure 6.3(a)), agents are initialized in the bottom right room and then move to the upper right room. The arrow represents the moving direction. The size of the gate is the parameter to evaluate. Each parameter has three value choices. Hence, the total number of clones is 27. The cloning tree of scenario 1 is shown in Figure 6.3(c). Both cloning strategies are applied to this scenario.

In the second scenario (Figure 6.3(b)), agents are initialized at two sides and cross the room to the other side. The arrows represent agent moving directions. Whether an obstacle exists at the specified locations is the parameter to be evaluated. The total number of clones is 4. The cloning tree of scenario 2 is shown in Figures 6.3(d) and 6.3(e), depending on whether agents of $C(2,1)$ senses p_2 first (Figure 6.3(d)) or agents of $C(1,2)$ senses p_1 first (Figure 6.3(e)). Because of the limitation of the top-down strategy as discussed in Section 6.2, it will not generate cloned simulation instance $C(2,2)$ correctly.

The verification experiment is performed on the CPU platform. Both the top-down and the bottom-up cloning strategies are verified with the full combinations of parameter value choices. To show the correctness of the two cloning strategies, the clone $C(3,3,3)$ of scenario 1 and two versions of clone $C(2,2)$ of scenario 2 at the bottom in each cloning tree are compared with their stand-alone simulation instances (i.e., instance that contains all agents at the beginning of simulation). These two clones in scenarios 1 and 2 respectively share most computations from their ancestor clones and are most likely to be erroneous. If these two clones are correct, then all the other clones can be inferred to execute correctly.

Figure 6.3: Scenarios and their corresponding cloning trees

The three figures in Figure 6.4 are the results of scenario 1. Both the top-down and the bottom-up strategies can correctly handle this scenario. There are 32 agents in this scenario. In Figure 6.4, each line represents the per-step position difference of an agent between the cloned simulation execution and the stand-alone simulation execution. Figure 6.4(a) compares the top-down cloning strategy (TD) and the stand-alone execution (SA), and Figure 6.4(b) compares the bottom-up cloning strategy (BU) and SA. The agents' initial positions are the same in all executions. As shown in the figures, the execution results (agent positions at each time step) of cloning strategies are identical to the results of the stand-alone execution.

To show that the scenario 1 cannot tolerate even a slight change, Figure 6.4(c) compares the stand-alone instance (SA) and its variation that adds 0.1 unit displacement to the x-coordinate of an agent's initial position (denoted as SA with disp). As shown in Figure 6.4(c), even a small difference introduced to one agent at the beginning leads

Figure 6.4: Correctness verification: Scenario 1

to huge deviation from the original execution. This shows that the proposed cloning strategy does not incur any error.

The four figures in Figure 6.5 are obtained with scenario 2. As shown in Figures 6.5(a) and (b), cloning $C(2,2)$ from either $C(2,1)$ or $C(1,2)$ with the top-down cloning strategy does not generate identical results as its stand-alone execution. However, the bottom-up cloning strategy can generate identical agent traces no matter $C(2,2)$ is cloned from $C(2,1)$ or $C(1,2)$, as shown in Figures 6.5(c) and 6.5(d). The bottom-up cloning strategy is more generic than the top-down cloning strategy.

6.5.2 Cloning vs. sequential stand-alone execution

The top-down and bottom-up cloning strategies are compared with the sequential stand-alone execution to demonstrate the performance improvement resulted from the computation sharing. The experiment is performed on the sequential CPU platform.

Since the top-down cloning strategy cannot correctly simulate scenario 2, only scenario 1 (i.e., agent passing through three gates) is used for comparison. 128 agents are initiated

Figure 6.5: Correctness verification: Scenario 2

at the beginning of each execution. The total number of clones is 27. Three execution strategies are compared, i.e., stand-alone executions of 27 clones (SA), the top-down cloning strategy (TD) and the bottom-up cloning strategy (BU). Figures 6.6(a) and 6.6(b) show the accumulated execution time and the per-step execution time respectively.

As shown in Figure 6.6(a), both cloning strategies perform better than stand-alone executions of 27 clones. When comparing the top-down and the bottom-up cloning strategies, the top-down strategy only checks agents in the **AgentPool** to perform cloning operations while the bottom-up strategy checks the **Context**. However, it does not incur significant overheads. Based on the experiment results, the bottom-up cloning strategy performs as efficiently as the top-down cloning strategy.

Figure 6.6(b) shows the average execution time of every ten steps of all clones. At the beginning stage of execution, the top-down and bottom-up cloning strategies are much more efficient than the stand-alone execution because the total number of agents in all clones is small. At the later stage, when most agents are cloned, performance of the top-down and bottom-up strategies becomes similar to that of the stand-alone execution.

Figure 6.6: Comparing cloning strategies with stand-alone execution

Figure 6.6(c) shows the average time of cloning operations of every ten steps and the time percentage of cloning in per-step execution. The top-down strategy checks agents in the **AgentPool** to make cloning decision. Cloning takes more time as the number of agents in the **AgentPool** increases. The bottom-up cloning strategy checks agents in the **Context**. In the scenarios in this experiment, the number of agents in the **Context** is fixed and higher than that in the **AgentPool**. Therefore, the cloning time of the bottom-up cloning strategy is higher than that in the top-down cloning strategy at the beginning. As more and more agents are cloned with the progress of the simulation, the cloning time of both strategies decreases. Thus, less cloning operations are needed. In general, the cloning only takes a small percentage in the total per-step execution time, as shown in Figure 6.6(c) (the lines denoted as TD percentage and BU percentage for top-down and bottom-up cloning strategies respectively). As the simulation progresses, the percentage of cloning decreases to less than 1% of per-step execution time.

Figure 6.7: Throttling mechanism

6.5.3 Throttling mechanism

The throttling mechanism is to prevent the unnecessary agent cloning, i.e., an agent interacting with a cloned agent does not necessarily mean its state will change. In this experiment, we further show that even when the threshold is set to zero, meaning cloned agent is only removed if it has exactly the same state as the original, throttling mechanism is still able to achieve performance improvement.

In this section, the experiment setup is the same as Section 6.5.2. As shown in Section 6.5.2, since both the bottom up cloning strategy and the top-down cloning strategy achieve similar performance in this experiment setup, only the bottom up cloning strategy is used to demonstrate the effect of the throttling mechanism. Two cases are compared: one is with the throttling mechanism disabled, and the other is with the mechanism enabled and the threshold is set to **zero**. Figure 6.7 shows the total number of agents in all clones and the accumulated execution time of the two cases. As shown in the figure, when the throttling mechanism is disabled, agents are rapidly cloned in a few time steps; whereas when the throttling mechanism is enabled, the agents are gradually cloned. Consequently, the accumulated execution time with throttling mechanism enabled is less than that with the mechanism disabled.

6.5.4 Bottom-up cloning tree evaluation

The MST heuristic is proposed to generate a cloning tree to increase computation sharing among clones and reduce the total execution time of a parameter exploration task.

Figure 6.8: Cloning tree evaluation

Scenario 1 is used as a test case to evaluate its performance. The total number of clones is 27. Four tree structures are compared: The tree generated by the MST heuristic, a flat tree (linking all other clones to a chosen root), a tall tree (clones are linked in chain), and three random trees. The random trees are spanning tree generated with Wilson’s loop-erased random walk algorithm [108]. The accumulated execution times are compared. The result of the random trees is obtained by averaging the results of the three random trees.

The experiment is performed on the sequential CPU platform. Figure 6.8(a) shows the accumulated execution time of all clones with four tree structures. The cloning tree generated with the MST heuristic takes the least time to finish. Figure 6.8(b) shows the per-step number of agents in 27 clones under four tree structures. Amongst all four cases, the parallel exploration using MST has the least number of agents. This indicates it performs best in terms of computation sharing. The MST heuristic performs better than other cloning trees because parent and child clones have least parameter differences. Therefore, a parent clone can share more agents with child clones.

6.5.5 Cloning vs. parallel stand-alone execution

Cloning strategy introduces synchronization barriers between a parent clone and its child clones. On parallel architectures, introducing a barrier between parent and child clones reduces the number of clones that can be processed concurrently. In contrast, all stand-alone instances can be executed in parallel. But, the number of clones that can be

Figure 6.9: Bottom-up cloning strategy vs. parallel stand-alone execution on GPU

concurrently supported by the hardware is limited. In this part of the experiment, the trade-off between computation sharing achieved by cloning strategy and concurrency achieved by parallel stand-alone execution is evaluated.

The experiment is performed on GPU. Scenario 1 is used for evaluation. The 8 clones marked in gray in Figure 6.3(c) are used in this part of experiment (this is equivalent to the situation where each parameter has only two value choices). As shown in Figure 6.9, when the number of agents in each clone is small (i.e., 32), the parallel stand-alone execution takes less time to finish (Figure 6.9(a)). This is because in the execution with the cloning strategy, the execution order of clones are based on the cloning tree, and synchronization barriers are set between each level of the tree. Thus, the 8 clones are divided into four batches at each time step (i.e., $\{C(1,1,1)\}$, $\{C(2,1,1), C(1,2,1), C(1,1,2)\}$, $\{C(2,2,1), C(2,1,2), C(1,2,2)\}$, and $\{C(2,2,2)\}$). In comparison, all 8 simulation instances are processed in one batch using the stand-alone parallel execution. Although the total number of agents processed with cloning strategy is less, parallel stand-alone execution performs better because all agents are concurrently processed in one batch.

When the number of agents in each clone is large (1024), the cloning strategy becomes beneficial (Figure 6.9(b)). Since the hardware cannot support the parallel execution of all agents, the agents queue for processing when all stand-alone instances are launched concurrently. The cloning strategy also cannot process all agents concurrently. But cloning reduces the total number of agents to process. Therefore, it uses less time to finish.

Figure 6.10: Example parameter setups of the large scenario

The parallelism supported by the hardware is usually limited. When simulating large scale scenarios, it is better to adopt the cloning strategy than simply running stand-alone simulation instances in parallel.

6.5.6 Large scale case studies

A large scale scenario is evaluated to show the effectiveness of the bottom-up cloning strategy. The scenario emulates a floor planing for a large office. There are 16 randomly generated candidate obstacles in the environment. A subset of obstacles will be picked for each simulation run. The experiment is to study how different combinations of obstacles may affect agent evacuation. The total number of combinations is 65536 (2^{16}). Three parameter value combinations are depicted in Figure 6.10. The squares are obstacles, and the dots are agents. Agents are evacuated through the nearest gate. There are eight gates in the environment. Agent movement is simulated using the social force model.

In this part of the experiment, 1000 out of the total 65536 combinations are selected to test the execution time of the cloning strategy. The MST heuristic is used to generate the cloning tree. The comparison is made between the bottom-up cloning strategy and the stand-alone instances executed on sequential CPU platform, multi-core CPU platform with OpenMP enabled, and GPU platform. In Figure 6.11, each line represents the accumulated execution time of the scenario consisting of 1000 clones. Because of the limitation of the top-down cloning strategy, it does not generate the correct simulation results and is therefore not included in the comparison.

Figure 6.11: Performance of the bottom-up cloning strategy on three platforms

As shown in the figure, cloning achieves great speedups in each platform. When comparing the execution time across platforms, GPU execution is much faster than the sequential CPU execution, but slower than the OpenMP CPU execution. This is because the workload of processing 1000 clones is too large for GPU to be processed completely in parallel. Therefore, part of the processing is serialized. In addition, the clock rate of GPU is much slower than that of CPU. Nevertheless, the experiment shows that the proposed cloning strategy is effective to speed up the execution of multiple ABS instances on various architectures and platforms.

6.6 Summary

In this chapter, a bottom-up cloning strategy is proposed. The Bottom-up cloning strategy overcomes the limitations of the top-down cloning strategy. The challenge to utilize the bottom-up cloning strategy is to construct a cloning tree that maximizes the computation sharing among simulation instances that evaluate the selected parameter value combinations. An MST heuristic comparing the parameter differences is proposed to construct the cloning tree. A large parameter exploration task consisting of 1000 clones was used to demonstrate the efficiency of the bottom-up cloning strategy on sequential CPU platform, multi-core CPU platform, and GPU platform. Experiment results show that the proposed cloning strategy can effectively speed up the execution of multiple ABS instances on various platforms.

Chapter 7

Conclusions and Future Work

This chapter concludes the thesis and discusses several directions for future work.

7.1 Conclusions

In summary, this thesis presents strategies to accelerate the execution of a single ABS instance and the execution of the parameter space exploration task that consists of multiple simulation instances.

First, to improve the performance of ABS, efficient execution strategies are proposed for executing two common and fundamental ABS modules on GPU, namely the agent management module and the agent interaction module. Regarding the agent management module, an **AgentPool** data structure and associated operations are particularly designed to support efficient agent creation and deletion of ABS. To improve the performance of the agent interaction module on GPU, an efficient neighbor searching execution strategy is proposed by carefully utilizing the GPU memory hierarchy, especially the shared memory. To demonstrate the performance of the proposed strategies, our strategies were compared with the state-of-the-art toolkits. The **AgentPool** was compared with CUDA native memory allocator and FLAME (an innovative ABS framework executed on GPU) to test dynamic agent creation and deletion. The proposed agent interaction module was also compared with the FLAME. The experiment results show the efficiency and effectiveness of the proposed **AgentPool** and neighbor interaction strategies.

Second, a library is proposed to support agent-based simulation on GPU, namely GSim. The main contribution is to provide a set of easy-to-use APIs to facilitate modeling and acceleration of ABS execution. The complex GPU programming required to

optimize ABS execution is hidden from GSim users. The workflow of GSim is customizable. The users can add application-specific logics to the GSim workflow. To show that our approach can accommodate a wide range of applications, four applications were implemented, including game-of-life, flocking boids, prey-and-predator, and the crowd simulation using social force. The performance of GSim was compared with FLAME and MASON. The experimental results show that, our GPU implementation always performs better than MASON when the number of agents is large. This is expected since GPU has much more processing elements than a multi-core CPU. When continuous space is used in the agent-based simulations, our implementation is also able to achieve better performance than the FLAME on GPU.

Besides the execution strategy for single simulation instance, incremental simulation cloning strategies are proposed to efficiently execute parameter space exploration tasks consisting of multiple simulation instances. Two incremental simulation cloning strategies are proposed, namely the top-down cloning strategy and the bottom-up cloning strategy.

As the third piece of work, a top-down cloning strategy is presented. Agent cloning in the top-down strategy is performed simultaneously with agent processing, that is, whenever an agent senses a parameter, it will be cloned. Its implementation on GPU is also discussed. Experiments were conducted using an agent-based evacuation simulation to demonstrate the effectiveness and efficiency of our approach. Compared to the stand-alone executions, the cloning approach achieves speedups.

Finally, as the fourth piece of work, two limitations of the top-down cloning strategy are identified, that is, the parameter value combinations evaluated by parent and child clones differ only at one parameter, and all agents need to sense parameters in the same sequence. To overcome these limitations, an improved cloning strategy is proposed, namely the bottom-up cloning strategy. The challenge to utilize the bottom-up cloning strategy is to construct a cloning tree that maximizes the computation sharing among simulation instances that evaluate selected parameter value combinations. An MST heuristic comparing the parameter differences is proposed to construct the cloning tree.

Case studies were conducted to verify the correctness of the two cloning strategies. Regarding the effectiveness, on sequential CPU platform, using both cloning strategies to execute multiple simulation instances can effectively reduce the overall execution time

compared with executing stand-alone simulation instances. On a parallel platform, the cloning strategy performs better than stand-alone executions when the scale is large and agents of all simulation instances cannot be concurrently processed. Finally, a large parameter exploration task consisting of 1000 clones was performed to demonstrate the efficiency of the bottom-up cloning strategy on sequential CPU platform, multi-core CPU platform, and GPU platform. On all platforms, the bottom-up cloning strategy achieves significant speedups.

In summary, our contributions in this research to the current state-of-the-art are: (1) efficient strategies to support two common ABS modules on GPU, namely the agent management module and agent interaction module; (2) a GSim library to facilitate the implementation and optimization of ABS applications on GPU; (3) an incremental ABS cloning strategy named the top-down cloning strategy which accelerates the execution of multiple ABS instances with different parameters but of the same model; and (4) a bottom-up cloning strategy which overcomes the limitations of the top-down cloning strategy to support broader range of parameter space exploration tasks.

7.2 Future Work

7.2.1 Enhancing the single ABS execution on GPU

First, the current strategies put more emphasis on the 2D continuous space environment. They are not optimized for ABS modeled using cell-based discrete environment. Efficient strategies for agents interacting in other environments, such as 3D continuous and network-based environment, need to be further proposed and optimized. The strategies for 2D continuous space environment can also be improved. For example, there are more efficient spatial indexing techniques, such as Z-indexing [69]. Adopting a better indexing strategy will allow agents corresponding to the threads in the same block to be geographically close to each other, thus further reducing irrelevant queries and enhancing the performance.

Second, the data structure of `AgentPool` can also be enhanced. In the current design, the `AgentPool` need to be given a fixed pool size at initiation. This is a limitation of current work. This limitation naturally brings out a future work to support automatic expanding and shrinking of the `AgentPool`.

Third, efficiency is a primary concern in this research. To increase the efficiency of executing a single simulation instance, multi-GPU execution strategies can be designed to exploit greater parallelism. The issues include but not limit to simulation partitioning, load balancing, and synchronization.

Fourth, more features should be provided to increase the usability of the GSim library. Except the agent management module and the agent interaction module, there are other modules that facilitate modeling, execution and analysis of ABS. There are modules that are commonly provided by the matured ABS frameworks (e.g., MASON, Repast, NetLogo), and GSim could be enriched to support these common modules.

7.2.2 Enhancing the ABS cloning

First, The simulation cloning techniques can be further enriched. The trade-offs between computation sharing and parallelism can be further investigated. Currently, all clones form a single cloning tree. In the future research, different tree structures can be analyzed. For example, clones can be divided into groups, and each group can form a tree. All clones can form a cloning forest that consists of multiple cloning trees. In this way, the clones can be processed in a parallel and distributed manner, i.e., each cloning tree in the cloning forest can be assigned to a computing node in the cluster and then the clones in the cloning tree can be executed on the computing node using the approaches proposed in this thesis. Besides, the way to estimate the weight coefficient of parameter difference w_i can be further investigated to guide the generation of cloning tree with high computation sharing.

Second, there are two ways to further speedup the exploration of multiple parameter configurations. The first way is to develop a framework that integrates the proposed simulation cloning mechanisms with evolutionary algorithms provided by an optimization engine (e.g., ECJ [62] or CASE [26]) so as to reuse the computations amongst simulation instances as well as to reduce the size of the parameter space to be explored. The second way is to investigate execution strategies based on more powerful architectures such as multi-GPU clusters to further speedup the execution.

Third, the experiments and case studies reported in this thesis are meant for the proof of concept, mainly used to demonstrate the effectiveness of the proposed strategies. To

thoroughly evaluate the proposed approaches, more realistic and complex test cases need to be implemented, for example, the study of parameter study of the relationship between the traffic light and traffic jam, and the study of wild fire spreading under different terrain conditions. Besides, the proposed cloning strategy is suitable for the independent parameters but not interrelated parameters. Issues related to the interrelated parameters should also be investigated.

Author's publications

- Journal publications:
 - **Xiaosong Li**, Wentong Cai, and Stephen John Turner. Supporting Efficient Execution of Continuous Space Agent-based Simulation on GPU. *Concurrency and Computation: Practice and Experience*, 28(12):3313-3332, 2016.
 - **Xiaosong Li**, Wentong Cai, and Stephen John Turner. Cloning Agent-based Simulation. To appear in *ACM Transactions on Modeling and Computer Simulation*.
- Conference publications:
 - **Xiaosong Li**, Wentong Cai, and Stephen John Turner. Cloning Agent-based Simulation on GPU. In *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pp. 173-182. 2015.
 - **Xiaosong Li**, Wentong Cai, and Stephen John Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of 2014 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 87-96. 2014.
 - Yan Xu, Gary Tan, **Xiaosong Li**, and Xiao Song. Mesoscopic traffic simulation on CPU/GPU. In *Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pp. 39-50. 2014.
 - **Xiaosong Li**, Wentong Cai, and Stephen John Turner. GPU accelerated three-stage execution model for event-parallel simulation. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pp. 57-66. 2013.

References

- [1] B. G. Aaby, K. S. Perumalla, and S. 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*, pages 29:1–29:10. ICST: Malaga, 2010.
- [2] S. J. Aarseth and S. J. Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.
- [3] M. Al-Zinati, F. Araujo, D. Kuiper, J. Valente, and R. Wenkster. Divas 4.0: A multi-agent based simulation framework. In *Proceedings of the 2013 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 105–114. IEEE, 2013.
- [4] C. Alexopoulos. A comprehensive review of methods for simulation output analysis. In *Proceedings of the 38th conference on Winter simulation*, pages 168–178. IEEE: Monterey, 2006.
- [5] W. B. Arthur, S. N. Durlauf, and D. A. Lane. *The economy as an evolving complex system II*, volume 28. Addison-Wesley Reading, MA, 1997.
- [6] R. Bagni, R. Berchi, and P. Carliello. A comparison of simulation models applied to epidemics. *Journal of Artificial Societies and Social Simulation*, 5(3), 2002.
- [7] J. Barceló et al. *Fundamentals of traffic simulation*, volume 145. Springer, 2010.
- [8] A. Bestavros and B. Wang. Multi-version speculative concurrency control with delayed commit. Boston University Computer Science Department, 1993. [Technical Report].

- [9] E. Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002.
- [10] B. Calvez and G. Hutzler. Parameter space exploration of agent-based models. In *Proceedings of the 9th international conference on Knowledge-Based Intelligent Information and Engineering Systems*, pages 633–639. Springer: Melbourne, 2005.
- [11] K. M. Carley, D. B. Fridsma, E. Casman, A. Yahja, N. Altman, L.-C. Chen, B. Kaminsky, and D. Nave. Biowar: scalable agent-based model of bioattacks. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 36(2):252–265, 2006.
- [12] A. Cayley. A theorem on trees. *Quarterly Journal of Mathematics*, 23(376-378):69, 1889.
- [13] D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low. Incremental HLA-based distributed simulation cloning. In *Proceedings of the 36th conference on Winter simulation*, pages 386–394. Winter Simulation Conference: Washington, DC, 2004.
- [14] D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low. Algorithms for HLA-based distributed simulation cloning. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(4):316–345, 2005.
- [15] D. Chen, S. J. Turner, B. P. Gan, and W. Cai. HLA-based distributed simulation cloning. In *Proceedings of 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 244–247. IEEE: Budapest, 2004.
- [16] D. Chen, S. J. Turner, B. P. Gan, W. Cai, and J. Wei. A decoupled federate architecture for distributed simulation cloning. In *Proceedings of the 15th European Simulation Symposium*, pages 131–140. SCS European Publishing House: Delft, 2003.
- [17] D. Chen, S. J. Turner, B. P. Gan, W. Cai, J. Wei, and N. Julka. Alternative solutions for distributed simulation cloning. *Simulation*, 79(5-6):299–315, 2003.

REFERENCES

- [18] D. Chen, L. Wang, A. Y. Zomaya, M. Dou, J. Chen, Z. Deng, and S. Hariri. Parallel simulation of complex evacuation scenarios with adaptive agent models. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):847–857, 2015.
- [19] F. Cicirelli, A. Furfaro, A. Giordano, and L. Nigro. Parallel simulation of multi-agent systems using terracotta. In *Proceedings of the 2010 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 219–222. IEEE, 2010.
- [20] T. M. Cioppa, T. W. Lucas, and S. M. Sanchez. Military applications of agent-based simulations. In *Proceedings of the 2004 Winter Simulation Conference*, volume 1. IEEE, 2004.
- [21] N. Collier and M. North. *Repast HPC: A platform for large-scale agent-based modeling*. Wiley, 2011.
- [22] J. Conway. The game of life. *Scientific American*, 223(4):120–123, 1970.
- [23] B. Craenen, P. Murgatroyd, G. Theodoropoulos, V. Gaffney, and V. Suryanarayanan. Mwgrid: A system for distributed agent-based simulation in the digital humanities. In *Proceedings of the 2012 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 124–131. IEEE, 2012.
- [24] D. DasGupta. *An overview of artificial immune systems and their applications*. Springer, 1999.
- [25] W. J. Davis. On-line simulation: Need and evolving research requirements. In *Handbook of simulation*, pages 465–516. Wiley-Interscience, 1998.
- [26] J. Decraene, M. Y.-H. Low, F. Zeng, S. Zhou, and W. Cai. Automated modeling and analysis of agent-based simulations using the CASE framework. In *Proceedings of 11th International Conference on Control Automation Robotics & Vision (ICARCV)*, pages 346–351. IEEE: Singapore, 2010.
- [27] D. D’Ambrosio, G. Filippone, R. Rongo, W. Spataro, and G. A. Trunfio. Cellular automata and gpgpu: an application to lava flow modeling. *International Journal of Grid and High Performance Computing (IJGHPC)*, 4(3):30–47, 2012.

REFERENCES

- [28] J. M. Epstein, D. M. Goedecke, F. Yu, R. J. Morris, D. K. Wagener, and G. V. Bobashev. Controlling pandemic flu: the value of international air travel restrictions. *PloS one*, 2(5):e401, 2007.
- [29] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient GPU implementation for large scale individual-based simulation of collective behavior. In *International Workshop on High Performance Computational Systems Biology*, pages 51–58. IEEE, 2009.
- [30] A. L. Espindola, D. Girardi, T. J. Penna, C. T. Bauch, A. S. Martinez, and B. C. Cabella. Exploration of the parameter space in an agent-based model of tuberculosis spread: emergence of drug resistance in developing vs developed countries. *International Journal of Modern Physics C*, 23(06):1–9, 2012.
- [31] M. J. Gibson, E. C. Keedwell, and D. A. Savić. An investigation of the efficient implementation of cellular automata on multi-core cpu and gpu hardware. *Journal of Parallel and Distributed Computing*, 77:11–25, 2015.
- [32] P. W. Glynn and P. Heidelberger. Analysis of parallel replicated simulations under a completion time constraint. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1(1):3–23, 1991.
- [33] A. P. Goldberg. Virtual time synchronization of replicated processes. In *Proceedings of the 6th Workshop on Parallel and distributed simulation*, pages 107–116. SCS Simulation Series: Newport Beach, 1992.
- [34] S. Green. Particle simulation using CUDA. NVIDIA: Santa Clara, CA., 2010. [Technical Report].
- [35] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 3. ACM, 2011.
- [36] M. Hanai, T. Suzumura, G. Theodoropoulos, and K. S. Perumalla. Exact-differential large-scale traffic simulation. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation*, pages 271–280. ACM, 2015.

REFERENCES

- [37] M. Hanai, T. Suzumura, G. Theodoropoulos, and K. S. Perumalla. Towards large-scale what-if traffic simulation with exact-differential simulation. In *Proceedings of the 2015 Winter Simulation Conference*. WSC, 2015.
- [38] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on GPUs. In *Proceedings of the 25th International Conference on Computer Graphics*, pages 63–70. SBC: Petropolis, 2007.
- [39] K. A. Hawick, A. Leist, and D. P. Playne. Regular lattice and small-world spin model simulations using CUDA and GPUs. *International Journal of Parallel Programming*, 39(2):183–201, 2011.
- [40] D. Helbing. Agent-based modeling. In *Social self-organization*, pages 25–70. Springer, 2012.
- [41] D. Helbing and A. Johansson. *Pedestrian, crowd and evacuation dynamics*. Springer, 2009.
- [42] D. Helbing and P. Molnar. Social force model for pedestrian dynamics. *Physical review E*, 51(5):4282, 1995.
- [43] A. Hérault, G. Bilotta, and R. A. Dalrymple. SPH on GPU with CUDA. *Journal of Hydraulic Research*, 48(S1):74–79, 2010.
- [44] R. R. Hill, R. Carl, and L. Champagne. Using agent simulation models to examine and investigate search theory against a historical case study. *Journal of Simulation*, 1(1):29–38, 2006.
- [45] C. Hommes. Interacting agents in finance. In *The New Palgrave Dictionary of Economics*. Palgrave Macmillan, 2008.
- [46] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE, 2011.

- [47] G. Horne and T. Meyer. Data farming: Discovering surprise. In *Proceedings of the 36th conference on Winter simulation*, pages 807–813. Winter Simulation Conference, 2004.
- [48] G. Horne and S. Seichter. Data farming in support of nato operations: methodology and proof-of-concept. In *Proceedings of the 2014 Winter Simulation Conference*, pages 2355–2363. IEEE, 2014.
- [49] M. Hybinette. Just-in-time cloning. In *Proceedings of the 18th Workshop on Parallel and distributed simulation*, pages 45–51. ACM, 2004.
- [50] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(4):378–407, 2001.
- [51] M. Hybinette and R. M. Fujimoto. Scalability of parallel simulation cloning. In *Proceedings of the Simulation Symposium*, pages 275–282. IEEE, 2002.
- [52] M. Hybinette, E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed. Sassy: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In *Proceedings of the 2006 Winter Simulation Conference*, pages 926–933. IEEE: Monterey, CA, 2006.
- [53] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. Simulation of information propagation over complex networks: Performance studies on multi-gpu. In *Proceedings of 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 179–188. IEEE, 2013.
- [54] A. Jnsm. Flocking with predator. <http://www.openprocessing.org/sketch/126516>, 2013. [Online, Accessed May 30, 2014].
- [55] E. Kiesling, M. Günther, C. Stummer, and L. M. Wakolbinger. Agent-based simulation of innovation diffusion: a review. *Central European Journal of Operations Research*, 20(2):183–230, 2012.
- [56] T. A. Kohler, G. J. Gumerman, and R. G. Reynolds. Simulating ancient societies. *Scientific American*, 293(1):76–84, 2005.

- [57] D. Król, M. Orzechowski, J. Kitowski, C. Niethammer, A. Sulisto, and A. Wafai. A cloud-based data farming platform for molecular dynamics simulations. In *Proceedings of 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 579–584. IEEE, 2014.
- [58] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle. Multi-level parallelism for time- and cost-efficient parallel discrete event simulation on gpus. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 23–32. IEEE: Zhang Jia Jie, 2012.
- [59] M. Lauren and R. Stephen. Map-aware non-uniform automata (MANA)-a new zealand approach to scenario modelling. *Journal of Battlefield Technology*, 5:27–31, 2002.
- [60] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe. Using gpu for multi-agent multi-scale simulations. In *Distributed Computing and Artificial Intelligence*, pages 197–204. Springer, 2012.
- [61] S. Luke. *Multiagent simulation And the MASON library*. George Mason University: Fairfax, VA, 2011.
- [62] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop. Ecj: A java-based evolutionary computation research system. <http://cs.gmu.edu/eclab/projects/ecj>. [Online. accessed on Nov, 2015].
- [63] L. Luo, S. Zhou, W. Cai, M. Y. H. Low, and M. Lees. Toward a generic framework for modeling human-like behaviors in crowd simulation. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 02*, pages 275 – 278. IET, 2009.
- [64] M. Lysenko and R. M. DSouza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):1–18, 2008.

- [65] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the Winter Simulation Conference*, pages 2–15. Winter Simulation Conference, 2005.
- [66] C. M. Macal, D. Sallach, and M. J. North. Emergent structures from trust relationships in supply chains. In *Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, pages 7–9. Argonne National Laboratory: Chicago, IL, 2004.
- [67] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Santa Fe Institute, Santa Fe, NM, USA, 1996.
- [68] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.
- [69] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.
- [70] D. Moser, A. Riener, K. Zia, and A. Ferscha. Comparing parallel simulation of social agents using cilk and opencl. In *Proceedings of 2011 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 88–97. IEEE, 2011.
- [71] M. J. North and C. M. Macal. *Managing business complexity: discovering strategic solutions with agent-based modeling and simulation*. Oxford University Press, 2007.
- [72] M. J. North, C. M. Macal, J. S. Aubin, P. Thimmapuram, M. Bragen, J. Hahn, J. Karr, N. Brigham, M. E. Lacy, and D. Hampton. Multiscale agent-based consumer market modeling. *Complexity*, 15(5):37–47, 2010.
- [73] M. J. North, E. Tatara, N. T. Collier, and J. Ozik. *Visual agent-based model development with Repast Symphony*. Argonne National Laboratory, Lemont, IL, USA, 2007.

- [74] NVIDIA. NVIDIA CUDA programming guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2013. [Online, Accessed May 13, 2014].
- [75] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.
- [76] H. Park and P. A. Fishwick. A GPU-based application framework supporting fast discrete-event simulation. *Simulation*, 86(10):613–628, 2010.
- [77] H. Park and P. A. Fishwick. An analysis of queuing network simulation using GPU-based hardware acceleration. *ACM Transactions on Modeling and Computer Simulation*, 21(3):18, 2011.
- [78] P. Pecher, M. Hunter, and R. Fujimoto. Efficient execution of replicated transportation simulations with uncertain vehicle trajectories. *Procedia Computer Science*, 51:2638–2647, 2015.
- [79] K. S. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced Discrete Simulation*, pages 74–81. IEEE, 2006.
- [80] K. S. Perumalla. Efficient execution on GPUs of field-based vehicular mobility models. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced Discrete Simulation*, pages 154–154. IEEE, 2008.
- [81] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *Proceedings of the 2008 Spring simulation multiconference*, pages 116–123. Society for Computer Simulation International, 2008.
- [82] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. GPU-based real-time execution of vehicular mobility models in large-scale road network scenarios. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced Discrete Simulation*, pages 95–103. IEEE, 2009.

- [83] M. J. Quinn, R. A. Metoyer, and K. Hunter-Zaworski. Parallel implementation of the social forces model. In *Proceedings of the 2nd International Conference in Pedestrian and Evacuation Dynamics*, pages 63–74. Fire Safety engineering group: London, 2003.
- [84] S. F. Railsback, S. L. Lytinen, and S. K. Jackson. Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623, 2006.
- [85] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *Proceedings of ACM SIGGRAPH Computer Graphics*, 21(4):25–34, 1987.
- [86] P. Richmond. Flame GPU technical report and user guide. Department of Automatic Control and Systems Engineering (ACSE), The University of Sheffield, Sheffield, UK, 2011. Technical Report.
- [87] P. Richmond, D. Walker, S. Coakley, and D. Romano. High performance cellular level agent-based simulation with flame for the gpu. *Briefings in bioinformatics*, 11(3):334–347, 2010.
- [88] A. Samanta, S. Hamel, and E. Schwegler. Order parameter aided phase space exploration under extreme conditions. *Bulletin of the American Physical Society*, 2016.
- [89] O. Shehory, K. Sycara, P. Chalasani, and S. Jha. Agent cloning: an approach to agent mobility and resource allocation. *IEEE Communications Magazine*, 36(7):58–63, 1998.
- [90] P.-O. Siebers, C. M. Macal, J. Garnett, D. Buxton, and M. Pidd. Discrete-event simulation is dead, long live agent-based simulation! *Journal of Simulation*, 4(3):204–210, 2010.
- [91] S. C. Silva, T. G. Carneiro, J. C. Lima, and R. R. Pereira. TerraME HPA: parallel simulation of multi-agent systems over SMPs. In *Proceedings of the ACM SIGSIM conference on Principles of Advanced Discrete simulation*, pages 361–366. ACM, 2013.

REFERENCES

- [92] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In *Proceedings of Innovative Parallel Computing*, pages 1–10. IEEE, 2012.
- [93] B. L. Stevens, F. L. Lewis, and E. N. Johnson. *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems*. John Wiley & Sons, 2015.
- [94] W. Tang and Y. Yao. A GPU-based discrete event simulation kernel. *Simulation*, 89(11):1335–1354, 2013.
- [95] T. Terano. Exploring the vast parameter space of multi-agent based simulation. In *Proceedings of the 2006 international conference on Multi-agent-based simulation*, pages 1–14. Springer: Hakodate, 2006.
- [96] T. Terano. Exploring the vast parameter space of multi-agent based simulation. In *Multi-agent-based simulation VII*, pages 1–14. Springer, 2007.
- [97] L. Tesfatsion and K. L. Judd. *Agent-based computational economics*, volume 2. Elsevier Amsterdam, 2006.
- [98] Y.-F. Tsai, F. Wang, Y. Xie, N. Vijaykrishnan, and M. J. Irwin. Design space exploration for 3-d cache. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(4):444–455, 2008.
- [99] K. Tumer and A. Agogino. Distributed agent-based air traffic flow management. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 255. ACM, 2007.
- [100] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference (GTC)*, volume 10, page 16. NVIDIA: San Jose, CA, 2010.
- [101] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.
- [102] B. Wagner, I. Sluijmers, D. Eichelberg, and P. Ackerman. Black-box reuse within frameworks based on visual programming. *Component-Based Software Engineering*, 10:57, 1998.

REFERENCES

- [103] M. Wagner, W. Cai, and M. H. Lees. Emergence by strategy: flocking boids and their fitness in relation to model complexity. In *Proceedings of the 2013 Winter Simulation Conference*, pages 1479–1490. IEEE: Washington, DC, 2013.
- [104] J. Wang, N. Rubin, H. Wu, and S. Yalamanchili. Accelerating simulation of agent-based models on heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 108–119. ACM, 2013.
- [105] F. Wende. *Simulation of spin models on NVIDIA graphics cards using CUDA*. Mathematisch-Naturwissenschaftliche Fakultät, 2010.
- [106] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 120–126. ACM, 2013.
- [107] U. Wilensky. *NetLogo 5.0.4 user manual*. Northwestern University: Evanston, IL, 2013.
- [108] D. B. Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the 28th ACM symposium on Theory of computing*, pages 296–303. ACM, 1996.
- [109] Y. Xie, G. H. Loh, B. Black, and K. Bernstein. Design space exploration for 3d architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(2):65–103, 2006.
- [110] C. Yang, I. Ono, S. Kurahashi, B. Jiang, and T. Terano. A grid based simulation environment for parallel exploring agent-based models with vast parameter space. In *Human Centered Computing*, pages 534–548. Springer, 2014.
- [111] C. Yang, I. Ono, S. Kurahashi, B. Jiang, and T. Terano. A grid based simulation environment for parallel exploring agent-based models with vast parameter space. In *Proceedings of First International Conference on Human Centered Computing*, pages 534–548. Springer, 2015.

- [112] D. Zehe, H. Aydt, M. Lees, and A. Knoll. Javascript distributed agent based discrete event simulation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 21–29. IEEE, 2013.
- [113] F. Zeng, J. Decraene, M. Y. H. Low, S. Zhou, and W. Cai. Evolving optimal and diversified military operational plans for computational red teaming. *Systems Journal*, 6(3):499–509, 2012.
- [114] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.
- [115] S. Zhou, D. Chen, W. Cai, L. Luo, M. Y. H. Low, F. Tian, V. S.-H. Tay, D. W. S. Ong, and B. D. Hamilton. Crowd modeling and simulation technologies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):20, 2010.