Efficient GPU Parallelization of the Agent-Based Models Using MASS CUDA Library

Elizaveta Kosiachenko

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2018

Committee:

Munehiro Fukuda, Committee Chair

Erika F. Parsons, Committee Member

Clark Olson, Committee Member

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Efficient GPU Parallelization of the Agent-Based Models Using MASS CUDA Library

Elizaveta Kosiachenko

Chair of the Supervisory Committee:

Professor Munehiro Fukuda

Computing & Software Systems

Agent-based models (ABMs) simulate the actions and interactions of autonomous agents and their effects on the system as a whole. Many disciplines benefit from using ABMs, such as biological systems modeling or traffic simulations. However, ABMs need computational scalability for practical simulation and thus consume a lot of time.

Multi-Agent Spatial Simulation (MASS) CUDA is a library, which allows using CUDA-enabled GPUs to perform multi-agent and spatial simulations efficiently while maintaining user-friendly and easily extensible API, which does not require the knowledge of CUDA on the user part. This thesis describes the optimization techniques for the spatial simulation, which allowed us to achieve up to 3.9 times speed-up compared to the sequential CPU execution of the same applications. We also propose solutions to challenges of implementing the support for dynamic agents as part of MASS CUDA library, including agent instantiation and mapping to the places, agent migration, agent replication and agent termination.

Table of contents

1. Introduction

This thesis presents the research of the optimization techniques for the spatial simulation and solutions to challenges of implementing the support for dynamic agents as part of GPU-based parallel computing library, including agent instantiation and mapping to the places, agent migration, agent replication and agent termination.

The three main contributions of this work to the ABM research community are: 1) applying three optimization techniques to make the spatial simulation on the GPU more efficient, 2) implementing the support for dynamic agents as part of MASS CUDA, including agent instantiation and mapping to the places, agent migration, agent replication and agent termination, while 3) maintaining the user-friendly API, which works with a variety of different applications and does not require an in-depth knowledge of CUDA programming.

1.1. Agent-Based Models (ABMs) Simulation

Agent-based modeling is a technique that is used to simulate the actions and interactions of autonomous agents and their effects on the system as a whole. An agent-based model (ABM) describes a system by representing it as a collection of dynamic agents often operating in a certain limited space. While each individual agent might have a fairly simple set of operations, the emergent behavior of a collection of these agents can be very complex. Such emergent behavior can be hard to predict or model with other techniques. Many disciplines benefit from using ABMs, such as biological systems modeling, disease spread prediction or traffic simulations. However, for practical application, ABMs often require large simulation involving thousands of agents. Such scale requires a lot of computational power and time.

1.2. CUDA Parallel Computing Platform

One way to provide computational scalability for ABMs is through the use of graphics processing units (GPUs). Initially designed for manipulating computer graphics and image processing, GPUs have evolved to be used in general-purpose computing. While average CPU might have 4 to 16 computing cores, many GPUs have thousands of processing cores. As a result, they can be very beneficial for the algorithms where the processing of large blocks of data is done in parallel.

Following the increased interest in general-purpose GPU computing, a lot of useful tools are being created for harnessing the parallel computing power of GPUs. One such tool is CUDA - a hardware/software platform developed by NVIDIA Corporation to allow GPUs to be used for high-performance computing. CUDA is currently the best-supported and most accessible platform for GPU-based parallel computing [32].

1.3. Opportunities and Challenges CUDA Platform Presents for ABM Simulations

Due to their high parallel processing power GPUs provide an opportunity to accelerate the agent-based models' execution. However, in order to create efficient GPU-based ABM simulations, the developers need to have a good understanding of the GPU programmability model and memory hierarchy, which are briefly outlined below [21].

GPU is composed of global memory (DRAM) and several stream multi-processors (SM) with multiple processing cores. Each SM can support hundreds of concurrent threads.

In CUDA, threads are organized into blocks, and all the blocks comprise a grid. Threads of the same block run on the same SM and are not separable. Each SM has limited resources such as shared memory and registers. Blocks assigned to the same SM have to compete for these resources. If a single block requests too many resources, the number of blocks that can be concurrently supported by an SM decreases, and the performance will be affected. Fortunately, the numbers of blocks and the number of threads per block are configurable by developers. If one block requests too many resources, one can always reduce the number of threads per block to reduce the resources requested by one block.

SM creates, manages, schedules, and executes groups of 32 parallel threads, called warps. Each warp of threads is executing in a batch manner. This represents the single-instruction-multiple threads (SIMT) parallel programming model utilized by GPU. Full efficiency is realized when all threads of a warp agree on their execution path. At every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction and issues the instruction to those threads. Execution context (program counters, registers, etc.) of each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of a warp. Therefore, switching execution between warps has no cost.

The memory of GPU has a hierarchical design as shown in Figure 1.1 and each type of memory has its use case. The global memory has the fewest number of limitations and can be accessed by all threads in all blocks. The amount of global memory on the device is large, but its speed of access is rather low. Data locality is very important for the effective bandwidth of the global memory loads and stores because the device coalesces global memory accesses issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth.

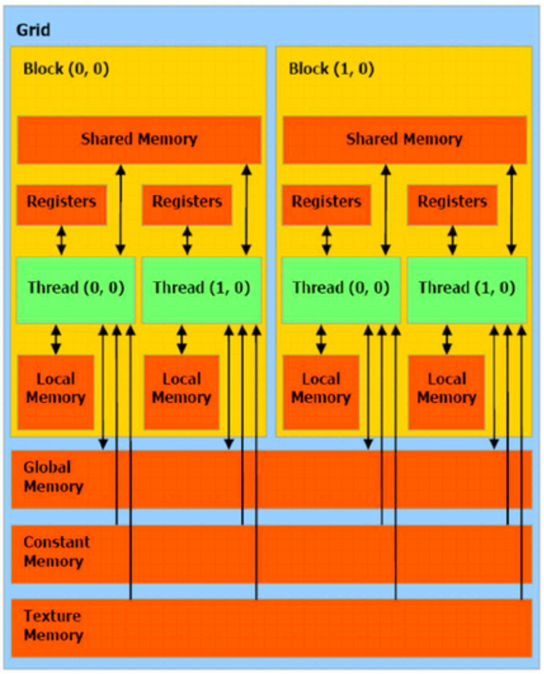

Figure 1.1: GPU memory hierarchy (from [23])

Constant and texture memory types are allocated on the GPU RAM as well, but are also cached on-chip during execution, so their effective speed is generally much higher than that of the global memory. The limitation for these memory types is that they are read-only.

Registers and local memory are only visible within a single thread that declares the variable. Register memory is very fast but limited in size. If the variable doesn't fit the register limitation in size, it's placed into a local memory, which is a part of the off-chip global device memory and thus has the same slow speed as the global memory type. Some on-chip caching is available for local and global memory, but the size of the cache is limited.

Another type of memory, which is widely utilized in various optimization techniques is shared memory. Only threads within one block can access the same shared memory address, but the speed of memory fetches is very high. Shared memory can be used as a user-managed cache, enabling higher bandwidth than is possible using global memory lookups.

Based on the GPU architecture and memory limitations, there are some ***specific challenges*** related to implementing agent-based models on the GPUs:

- Memory access patterns:
    - The straightforward implementation of agent replication and termination requires frequent dynamic memory allocation, which can be a severe bottleneck on the GPU, as it requires global synchronization of the device and thus stall of all the executing warps;
    - Search for neighboring agents can be inefficient on the GPU, as neighbors can be situated anywhere in the global memory and thus memory access does not coalesce;
- Branch divergence:
    - Agent-based models often include agents with different rules of behavior, which have different execution paths within a kernel. This results in thread divergence and thus reduces parallelism.

To overcome these challenges and implement an efficient agent-based model running on GPU users need an in-depth knowledge of CUDA and relevant optimization techniques. This can be a limitation to subject matter experts, who want to port their applications to GPU, but not necessarily possess the required GPU-programming skills.

1.4. MASS CUDA Library Concept

The main goal behind MASS CUDA library is to hide the CUDA implementation details from users and allow subject matter experts to create efficient agent-based simulations without needing to engage in CUDA programming, yet still, reap the performance benefits of parallel processing.

Similarly to MASS Java and MASS C++ [1], MASS CUDA library uses two important concepts: Agent and Place. Place objects represent locations and remain static during the whole simulation process. Agent instances execute different instructions, migrate between Places, can spawn new child Agent instances, or get terminated.

1.5. Current Problems in MASS CUDA

The first version of MASS CUDA was built in 2015 as part of the thesis project of Hart, N. B. "MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks" [2]. That version of the library implemented the encapsulation of the details of GPU parallel programming and provided a user-friendly API that did not require any CUDA knowledge from the user. However, the library also had several problems. One issue was poor performance - the resulting execution time of the system was 19% to 54% slower compared to the sequential version of the program depending on the problem size. Furthermore, that version of the library only supported spatial simulation and did not implement support of dynamic agents.

1.6. Research Goals and Scope

Based on the problems identified in the previous version of the MASS CUDA library, this thesis has two main goals:

1) research, implement and evaluate the techniques for optimizing the performance of the spatial simulation
2) research techniques for support of dynamic agents in ABMs, in particular, such functions as agent instantiation and mapping to space, agent migration, agent replication, and agent termination, and implement them as part of MASS CUDA library

While implementing new features and optimization techniques, we aim to follow the existing specification and API for MASS CUDA library as much as possible and limiting the number of CUDA-specific details exposed, so that the users of the library are not required to have preliminary GPU programming knowledge.

A benchmarking application was developed to measure the impact of the implemented performance optimization techniques. The selected application is SugarScape, which simulates the behavior of ants in the presence of sugar.

2. Related work

2.1. GPU Agent-Based Models, Frameworks and Techniques

There is a significant amount of research in agent-based models' optimization using GPU (and CUDA in particular). However, a lot of works are focusing on optimizing specific problems and not developing a framework suitable for a variety of problems. Furthermore, those simulations require users to have quite an in-depth knowledge of GPU programmability model and CUDA language. In the following subsections, we review the work that has been conducted in relation to application-specific ABMs' optimization, individual techniques for ABM simulations implementation as well as the few frameworks developed for optimizing agent-based models.

Application-specific ABMs utilizing GPU

There are a number of application-specific implementations of agent-based models on GPU that proved to provide good performance results and significant speed-up compared to the sequential execution. Some examples of research papers grouped by the application domain:

1) *Biology & Medicine*:
    a) Tuberculosis epidemic simulation [11]: identifying mechanisms of granuloma formation through the interaction of T-Cells and Macrophages represented as agents;
    b) Modeling of blood coagulation system [10]: simulating complex interactions involved in blood coagulation system involving reactants, enzymes, and products of the coagulation system.
    a) Systemic inflammatory response simulation [4]: generating population-level behaviors among the circulating blood and the EC cells lining the blood vessels.
    b) Protein structure prediction [8]: exploring the folding of different parts of a protein by concurrent agents.
2) *Physics*:
    a) Molecular dynamics simulation [20]: studying of the metal solidification process through the interaction of aluminum atoms.
3) *Mathematics*:
    a) Graph theory [9]: using Ant Colony Optimization algorithm to find an optimal path in a graph (Traveling Salesman Problem);
    b) Cooperative Particle swarm optimization [18]: stochastic optimization algorithm letting all the particles reach the global best in cooperative fashion through information exchange.
4) *Social Studies*:
    a) Traffic simulation: micro-simulation involving large groups of vehicle agents moving over the traffic network [30, 33, 34, 37], and traffic signal timing optimization [31];
    b) Crowd simulation and path planning: simulating large crowds of complex agents [36], navigating agents among moving obstacles [5], agent path planning on grid maps [6, 7] or arbitrary surfaces [35].

c) Collective motion simulation: studying the collective behaviors of self-propelled biological organisms, such as birds [16, 17] fish [13], or general flocking “boids” (individuals in animal crowd simulations) [24, 25] by considering the very large number of interactions among group members.

Some of these research papers describe straightforward implementations of the respective simulations on GPU; others provide optimization techniques that can also be applied to a wider range of simulations.

The main challenges identified as part of these works are the complexity of implementing them while supporting applications of various sizes and dimensionalities; being very application-specific and not extensible to the broader range of applications; decreased parallelism and overhead of constant memory transfers between CPU and GPU.

One of the techniques introduced in these works is the use of the tiling algorithm. It uses fast shared memory of the GPU to load “tiles” of the total grid of places, which are then executed within one block of threads. The “tiles” periodically exchange data dependencies between each other. This technique proves to demonstrate good performance results in the research by Cecilia et al. and Husselmann et al. [9][17]. However, when applied to a library, which is intended to support simulations of different sizes and dimensionalities, use of such algorithm would make the code extremely complex and hard to maintain or modify by the future contributors.

Another technique, which we considered not applicable to MASS CUDA, is the use of a hybrid GPU/CPU model described in the work by Li, D. et al. [20]. Separation of functionality between CPU and GPU is very application-specific, so it is not an option for the library, which should support a variety of different ABM simulations. Additionally, leaving part of the simulation on the CPU can potentially lead to decreased parallelism and overhead of memory transfers between CPU and GPU.

Several works [5][9] introduced custom thread-safe random number generation, which decreases the penalty of using the slow “cuRAND” library. This technique can be useful for a big range of ABMs, however in MASS CUDA random number generation is used only at initial agent allocation for spreading agents across space and thus can be performed on CPU before agent objects are created on GPU.

Some works introduced non-trivial ways to maintain a list of agents residing in a particular place. Research by Chen, W. et al.[10] demonstrates a way to store agents in two-directional linked lists, while the work of Strippgen, D. et al. [33] suggests storing agents in place in a dynamic FIFO queue. Those techniques are interesting to experiment with, but because their benefits to performance are non-obvious, we left those experiments beyond the scope of this research.

Some of the techniques for agent replication and garbage collection include processing agent replication in batches using a single thread to avoid the need for synchronization [10] and sorting agents by dead/alive for memory reuse[11]. Implementation of these techniques was left out of the scope of this research.

The techniques that did find application in our library include storing reusable parameters for GPU kernels in constant memory [16], use of “pragma unroll” for maximizing the use of register memory [16] (after experimentation we omitted this technique from the latest version due to the lack of performance improvement), allocating upper limit of memory for data structures to avoid dynamic memory allocation on the fly[33].

Despite the usefulness of some of the techniques in these works, all of them focus on specific simulations and thus might not be applicable to other simulations/domains. Furthermore, all of these works require users to use CUDA language to implement the specified simulations and techniques, while one of the main requirements for our project is to hide CUDA implementation details from the user and provide user-friendly API that does not require the in-depth knowledge of GPU programmability model.

Works focusing on specific techniques for implementing ABMs on GPU

Another category of research is focusing on specific techniques that might improve the performance of agent-based models. These techniques are intended to be applicable to various ABMs, not just one specific simulation.

The work by Aaby et al.[3] describes the latency-hiding mechanism for ABM simulation on GPU, where the grid of agents is separated into blocks allotted to independent processing elements. The blocks then exchange the data dependencies with neighboring blocks after updates in agent states. This approach is quite efficient as it takes advantage of the fast shared memory, which can be accessed by threads in one block. However, the use of such algorithm to support simulations of different sizes and dimensionalities makes the code extremely complex and hard to maintain or modify.

Other works by Hermellin et al.[14, 15] presents the GPU environmental delegation approach, which implies making a clear separation between the agent behaviors, managed by the CPU, and environmental dynamics, handled by the GPU. This kind of hybrid approach can better accommodate very dynamic systems, where agents evolve greatly in the course of the simulation, however leaving part of the simulation on the CPU can potentially decrease parallelism and introduce an overhead of memory transfers between CPU and GPU.

Lastly, the research by Li et al.[21] proposes an AgentPool data structure to handle agent creation and deletion on GPU. An instance of AgentPool is initiated with a fixed capacity equal to the maximum number of agents in the system, and the actual number of agents is tracked at runtime. The management of the pool is performed by manipulating pointers to reduce the overhead of dynamic memory allocation/deallocation on the device. This is the approach we adopted in our research as well.

Similarly to simulation-specific ABM implementations, most of the research projects focusing on specific techniques don't address the goal of providing user-friendly API that does not require in-depth knowledge of CUDA programming language.

Agent-based modeling frameworks utilizing GPU

A few frameworks are available in the area of ABM frameworks utilizing CUDA-enabled GPUs: FLAME GPU[26-29], Turtlekit[22] and MCMAS[19].

FLAME GPU is a high performance parallel framework for agent-based modeling, which is using an XML schema to represent agents, their functions, fields, messages, and memory allocations. The framework has certain limitations, in particular agents field types are limited to integer, float, and double data types, and framework use requires XML scripting skills in addition to the C programming language. Additionally, it does not provide an execution environment the agents can interact with.

Turtlekit is a Logo-based spatial ABM platform, implemented with Java. Its long-term goal is to develop a library of GPU modules implementing various environment dynamics specifically designed for spatial ABMs. Right now it implements the delegation of some of the agents' behaviors onto GPU. Because the library takes the hybrid approach of a partial delegation of simulation to GPU and because it does not yet implement the full range of possible agent behaviors in its modules, the amount of computation speed-up it can achieve through parallelism is limited.

MCMAS is a library intended to facilitate the implementation of agent-based models on GPU and many-core architectures through providing a set of commonly used functions and data structures. One of the limitations of this library is that the user is limited to the set of implemented functions, algorithms and data structures, which can be extended only through writing low-level GPU code. Another disadvantage is, similarly to TurtleKit, the use of the hybrid model with only a partial delegation of simulation to GPU, which potentially decreases parallelism and requires frequent memory transfers between CPU and GPU.

So, while the abovementioned frameworks address the issue of programmability, they still have some challenges remaining: not easily extendable API and, in case of Turtlekit and MCMAS, limited parallelism due to the use of the hybrid CPU/GPU model approach, which potentially decreases parallelism and requires frequent memory transfers between CPU and GPU.

We address the challenges of the aforementioned frameworks in MASS CUDA by providing an easy API, which does not require the knowledge of low-level GPU languages and is easy to extend by writing functions in C++. Additionally, the simulation is performed entirely on GPU, which allows to minimize memory transfers between CPU and GPU and maximize the parallelism in the system.

2.2. Previous Work on MASS CUDA

While this work is the first to provide full support for dynamic agents as part of MASS CUDA, it builds on top of thesis project of Hart, N. B. "MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks". The work describes the architecture and implementation techniques for the library, which allow to abstract away the CUDA implementation details from the user and achieve the APIs which are very similar to other versions of the MASS library: MASS C++ and MASS Java.

The resulting library completely hid CUDA implementation details from the user with several exceptions:

- The project must be compiled using nvcc (Nvidia CUDA compiler) with the required flags/options;
- All files that are normally .cpp are .cu;

- Functions in user-defined Place or Agent classes should be prepended with the macro MASS_FUNCTION (stands for __host__ __device__), which enables compiling of both host and device code.

In the current work we retained the same API as in the Harts thesis work and also took advantage of the library architectural model and split of library object classes (Place, Agent, and user classes derived from them) into behavior and state classes.

The architectural model proposed and implemented in that work is Model-View-Presenter. View represents the API of the library; Model represents the data model on both GPU and CPU; and Presenter represents the dispatcher coordinating the interaction of Model and View, performing memory transfers between CPU and GPU, and launching GPU kernels (functions executed on GPU). The architecture is shown in Figure 2.1.

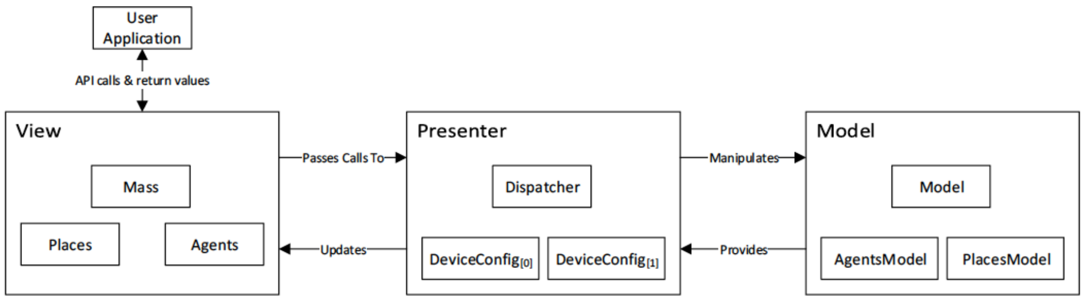

Figure 2.1: High-level architecture of the MASS CUDA library [2]

We decided to reuse the Model-View-Presenter model in the current work because it allows hiding the details of CUDA implementation from the user. It also encapsulates the GPU-specific implementation details into a separate Model component, which allows for easier code maintenance and modification.

Another technical decision that we reused from Harts work is splitting behavior and state into two different classes, where the behavior class, Agent or Place, contains all the functions as well as a single pointer to a generic AgentState and PlaceState class that contains all the fields. An array of concrete behavior classes are instantiated both on the host (CPU) and device(GPU), as well as an array of state classes of equal size. Each behavior instance is assigned a unique state instance. Memory transfer is achieved by copying the state array from host to device and back again.

Using this pattern allows for the creation of derived classes by a user which would extend Agent and Place classes and at the same time to manage all of the memory allocations and transfers between host and device behind the scenes.

While the primary goal of Harts work - to implement the encapsulation of the details of GPU parallel programming - was accomplished by his research, the resulting library did not achieve the performance goals as it showed a performance slowdown when compared to sequential computation of an identical simulation - the resulting performance of the system was 19% to 54% slower depending on the problem size. Furthermore, his research focused on spatial simulation and did not address the challenges of implementing dynamic agents instantiation, migration, termination and replication.

In the current work we reuse the API, architectural model and some of the technical decisions outlined above from the Harts thesis, and address some of its shortcomings, in particular improving the performance of the spatial simulation and addressing the challenges pertaining to the support for dynamic agents as part of the library: agents instantiation, mapping to space, migration, replication and termination.

2.3. New Research Contribution to MASS CUDA and GPU Agent-Based Modeling

This thesis research contribution is three-fold: 1) applying three optimization techniques to make the spatial simulation on the GPU efficient, 2) implementing the support for dynamic agents as part of MASS CUDA, including agent instantiation and mapping to places, agent migration, agent replication and agent termination, while 3) maintaining the user-friendly API, which works with a variety of different applications, does not require an in-depth knowledge of GPU programming and can be easily extended using just C++.

Compared to the previous version of the MASS CUDA library we also simplified it (decreased the number of lines of code by 32%, while simultaneously adding support for dynamic agents and improving the library performance). We also made some smaller improvements to the library functionality, for example, in case there are several GPUs present on the computer, the library now selects the GPU device with the highest compute capability instead of hard-coded values as it was implemented previously. This makes the process of porting the library to a new computer easier.

3. Performance Optimization for Spatial Simulation

While the goal of encapsulating the details of GPU parallel programming has been accomplished by the previous version of MASS CUDA[2], the resulting library did not achieve the performance goals as it showed a performance slowdown when compared to the sequential computation of an identical simulation.

To achieve better performance for the spatial simulation, we implemented and run performance measurements of several optimization techniques:

- use of constant memory;
- avoiding context switch between GPU kernels;

- selecting the optimal launch configuration of GPU kernels;
- using the "pragma unroll" compiler directive for loop unrolling;
- replacement of getter and setter function in the object classes with direct member access.

We also considered implementing the tiling or stencil algorithm but abandoned this techniques technique due to maintainability issues.

To measure the influence of the implemented techniques on the overall performance of the library spatial functionality we used the benchmarking application Heat2D, which simulates heat transfer in a mass of metal following the Euler method. The specifications of the hardware used in all of the experiments are described in section 5.1.Evaluation Environment.

Use of constant memory

As previously mentioned, constant memory is a relatively fast kind of GPU memory, which is initially allocated on the GPU RAM but is also cached during execution. This type of memory has a limitation of being read-only, so its use is limited to parameters that do not dynamically change throughout the simulation. In case of MASS CUDA, one such type of parameter identified is the array holding the relative coordinates of the neighbors that is used for data exchange between Place objects.

The transfer of even such a small amount of data (array of neighbor coordinates) from global to constant memory led to a noticeable performance improvement of 4% for the whole simulation.

Avoiding context switch between GPU kernels

In CUDA the kernel execution context (registers, cache, etc.) is maintained throughout the whole life of a kernel, but all the cached data is discarded as soon as the kernel execution is complete. Combining several GPU kernels operating on the overlapping data can allow to reuse the cached information and decrease the overall number of memory requests.

In MASS CUDA library one such combination of functions is exchangeAll() and the following callAll(). exchangeAll() function in the MASS CUDA library is used to collect data from the provided array of neighbors and save it into the message array in place. That data is then used by the subsequent functions invocations. And callAll() is used to execute a specific user-defined function on all the places objects in a collection. A common use pattern for these functions is the call to the exchangeAll() to collect some data from the neighboring places followed by the callAll() to use the collected data in some sort of computation.

We implemented an additional version of exchangeAll() which combines the exchangeAll() and callAll() functionalities in one GPU kernel call. The user can specify a function ID that will be executed for all Place objects after data collection is complete. The specified function is executed in the same GPU kernel that collects the data, so we avoid a context switch between the data collection kernel and calculation kernel, and thus the cached data can be reused for the calculation of results in the function that requires this data.

Implementing the modified exchangeAll() function resulted in an improvement of performance for the Heat2D simulation by 33% compared to the original MASS CUDA library.

Selecting the optimal launch configuration of GPU kernels

When making invocations of kernels on GPU developers need to specify the launch configuration of kernels, in particular, the number of blocks and number of threads per each block. As mentioned earlier, threads of the same block share limited resources such as shared memory, registers, and caches. If a single block requests too many resources, the number of blocks that can be concurrently supported by a GPU stream multiprocessor decreases, and the performance is negatively affected. Choosing the optimal launch configuration depends on the types and amount of memory utilization by the program and the specifications of the GPU device the application is launched on.

The previous version of MASS CUDA used the configuration of 512 threads per block. We ran a series of experiments using the Heat2D application to find out if this launch configuration is optimal. We used varying settings for the number of threads per block. The number of blocks in a grid was calculated as the total number of Places in the simulation divided by the number of threads per block selected, rounded up.

As shown in Figures 3.1 and 3.2, 512 threads per block was significantly slower than the optimal configuration for the Heat2D application implemented on top of MASS CUDA. The best configuration was 24 threads per block. Setting launch configuration parameters to the optimal values resulted in the 32% performance improvement compared to the original MASS CUDA library for the biggest simulation size we tested (1000 x 1000 places).

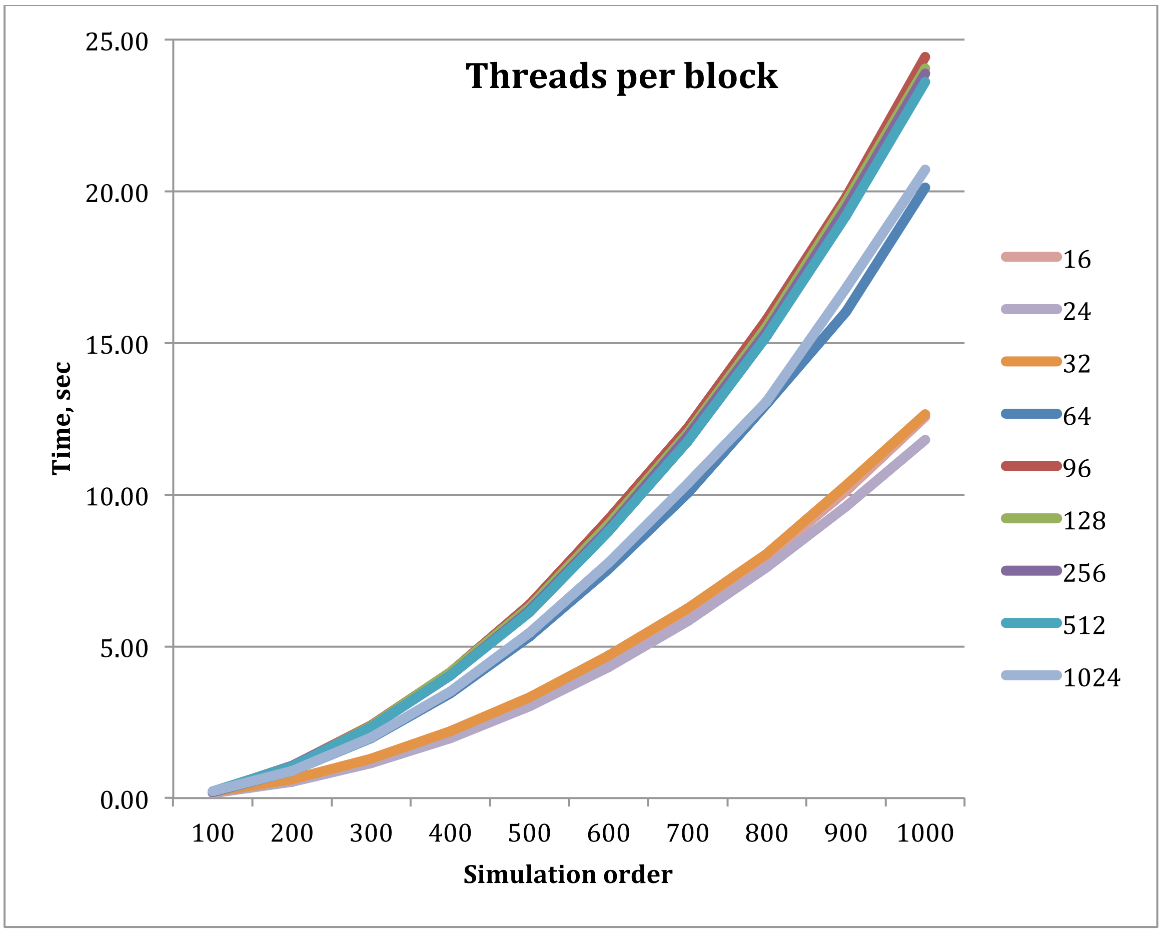

Figure 3.1: Performance by number of threads per block of the Heat2D spatial simulation implemented on top of MASS CUDA library

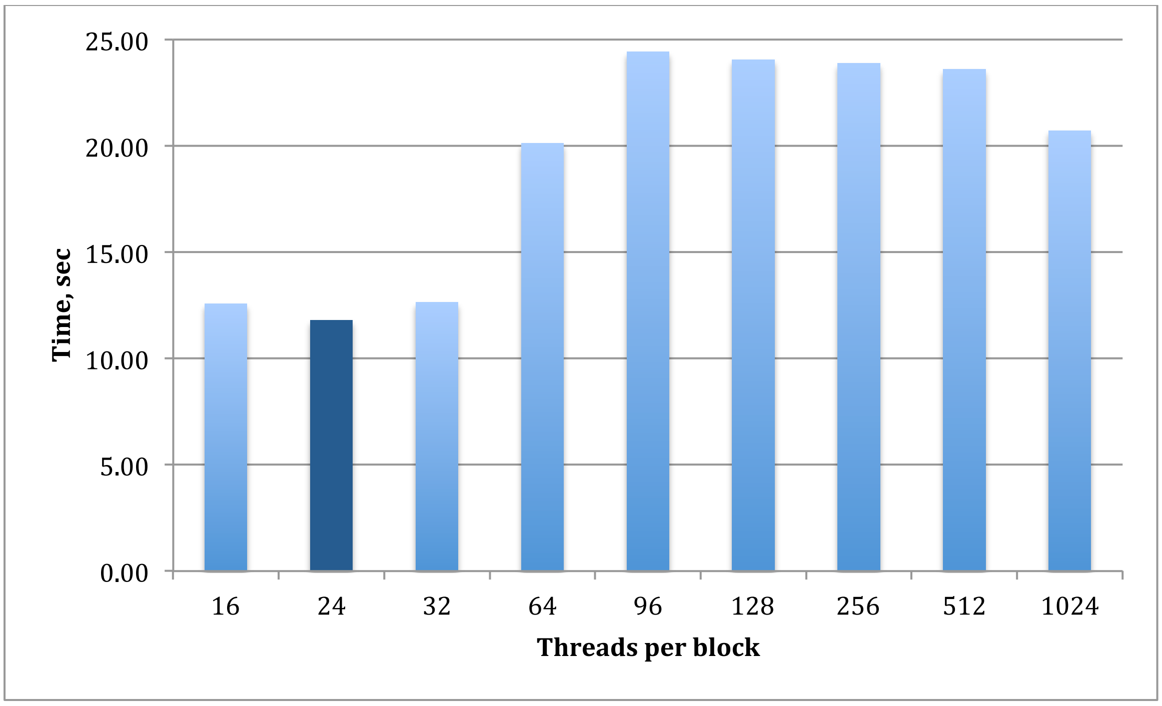

Figure 3.2: Performance by number of threads per block of the Heat2D spatial simulation of size 1000x1000 implemented on top of MASS CUDA library

The explanation for such a low optimal number of threads per block can be the high utilization of per-block resources by the MASS CUDA simulations, such as registers and caches. The Places memory space allocated to each thread is large. Therefore, a smaller number of threads per block can fit their Places memory space in the limited cache memory and have higher cache hit ratio.

The relevant profiling metrics of the most computation-heavy kernel in the MASS CUDA simulation – *mass::exchangeAllPlacesKernel()* - are shown in Table 3.1. According to the metrics, even though the achieved occupancy (ratio of the active warps per cycle to the maximum number of warps supported on a multiprocessor) of the 24 threads per block configuration is lower than the 512 threads per block configuration, its memory utilization is more efficient: L2 cache hit rate is much higher and the overall number of GPU DRAM accesses is significantly lower.

GPU kernel		**Importance (% of total GPU compute time)**	**Avg. Duration (ms)**	**Achieved Occupan-cy**	**L2 Hit Rate**	**Device Memory (DRAM) Read Transactions**	**Device Memory (DRAM) Write Transactions**
mass:: exchangeAll PlacesKernel()	at **24** threads per block	69%	4.26	25%	77%	6.3 M	3.0 M
	at **512** threads per block	87%	9.12	69%	17%	20 980.0 M	5.1 M

Table 3.1: Profiling results for the *mass::exchangeAllPlacesKernel()* of the MASS CUDA library executing the Heat2D application of size 1000x1000

Similar experiments were conducted for a different application - SugarScape, which simulates the behavior of ant colony in the presence of sugar and, in addition to spatial simulation, also includes dynamic agents. Figure 3.3 gives the best setting for the SugarScape application of 24 threads per block, similar to Heat2D. So, we can reasonably assume that this launch configuration of kernels might be optimal or close to optimal for a wider variety of applications implemented on top of MASS CUDA as well.

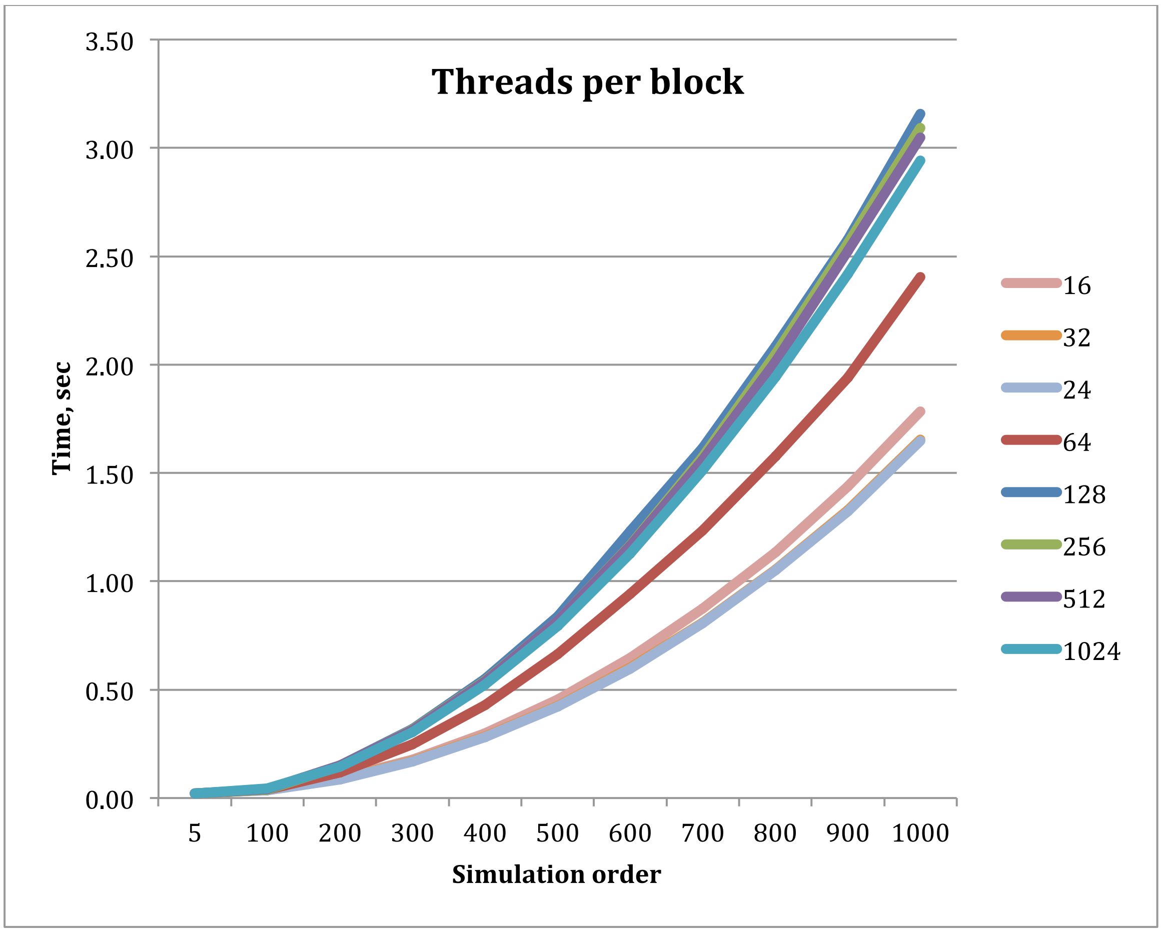

Figure 3.3: Performance by number of threads per block of the SugarScape dynamic agent simulation implemented on top of MASS CUDA library

Other techniques considered

As part of the process of improving MASS CUDA library spatial simulation performance, we considered several optimization techniques, which did not provide an expected boost in performance.

One of such techniques was using the "pragma unroll" compiler optimization directive, as mentioned in the work by Hidayat et al. [16]. This directive is intended to unroll loops in the GPU code at the compilation time in order to maximize the usage of the fast register memory and decrease the processor load. However, upon implementing this compiler directive in the most computation-heavy kernel in our library, we did not see any noticeable results in performance. This lack of improvement can be explained by the fact that most of the loops in our kernels were relatively small (below 10 iterations) and the number of iterations was known at compile time, so the compiler automatically performed this optimization, according to CUDA programming guide [23].

Another technique we tried was the replacement of getter and setter function in the object classes with direct member access to the respective field. We hoped that this change might improve memory access patterns within GPU kernels; however, it did not influence the performance. The explanation for this can be either the automatic optimization of such functions by the compiler or that these functions, in fact, do not influence GPU memory access patterns and their overhead is negligible. Upon finishing the experiment, we returned to the use of getter and setter functions, as they promote better coding style through encapsulation.

During the implementation of the improved MASS CUDA library, we also considered a technique mentioned in the works of Cecilia et al. and Husselmann et al.[9][17] - using tiling or stencil pattern to take advantage of the fast shared memory, accessible by threads in the same block. As mentioned earlier, the tiling technique uses fast shared memory of the GPU to load "tiles" of the total grid of places, which are then executed within one block of threads. During each iteration of the simulation "tiles" exchange data dependencies between each other. The stencil pattern works in a similar manner but includes the bordering places of the neighboring tiles into the cached data block. However, because MASS CUDA library is intended to support applications of different sizes and dimensionalities, use of such algorithms makes the code for the exchange of data dependencies between the "tiles"(blocks of threads) very convoluted. So we abandoned this optimization technique in favor of code maintainability.

Techniques' contribution to overall library performance improvement (based on the

Heat2D simulation of size 1000 by 1000) is shown in Figure 3.4. The biggest contribution was provided through optimizing the number of threads per block for kernel launches. Second-best was avoiding context switch between GPU kernels through combining exchangeAll() and callAll() functions to execute in one GPU kernel call. And the smallest, but still significant contribution was made by the use of constant memory for storing kernel parameters.

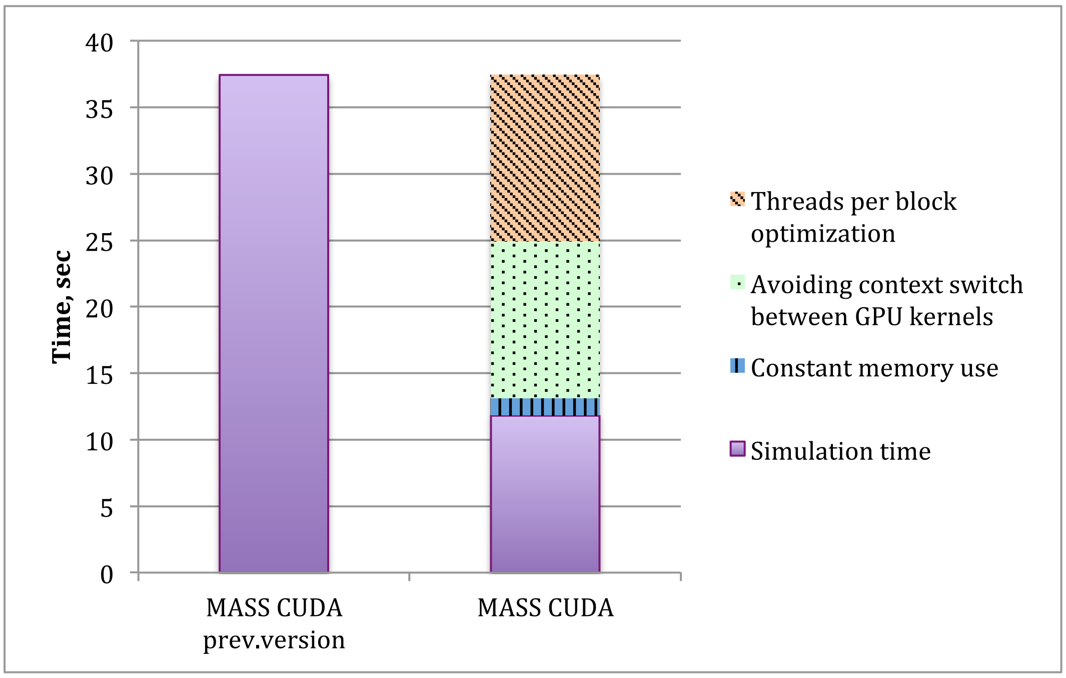

Figure 3.4: Contribution of different performance optimization techniques to the overall speed-up of the spatial simulation (Heat2D simulation of size 1000x1000)

To sum up, the 3.2 speed-up compared to the previous version of the library was achieved with three major techniques: the use of constant memory to store data frequently accessed by threads in different blocks, avoiding context switch between GPU kernels, and selecting the optimal launch configuration of kernels. The techniques that did not provide any performance improvement in MASS CUDA execution were using the “pragma unroll” compiler directive for loop unrolling and replacement of getter and setter function in the object classes with direct member access. And the technique that we did not select due to maintainability issues was the tiling algorithm.

4. Agent-Based Simulation Support in MASS CUDA

An essential goal of our research was to implement the support of dynamic agents as part MASS CUDA library, in particular, such functions as agent instantiation and mapping to space, agent migration, agent replication, and agent termination. In this chapter, we discuss the challenges we faced implementing this functionality and the technical decisions we made during the implementation process. We also go into more detail on three different approaches for conflict resolution during agent migration stage: (1) no deterministic migration conflict resolution; (2) migration conflict resolution using CUDA atomic functions; and (3) our custom algorithm for migration conflict resolution involving maintaining an array of all the Agents trying to migrate to a Place as part of that Place.

The general workflow of working with dynamic agents in MASS CUDA includes the following operations:

- Creation of the agent collection and mapping Agents to Places using the createAgents() function;
- Calling various functions on Agent objects using the callAll() function, which invokes the same user-defined method on all of the Agents in a collection;
- Calling the manageAll() function on the Agents collection to perform the termination, migration or replication of agent instances after invocation of the terminateAgent(), migrateAgent () or spawn() functions in any of the agents' functions.

4.1. Agent Instantiation and Mapping to Places

In MASS CUDA agents are instantiated across a grid of places. So, before creating the collection of Agents, the user needs to instantiate the MASS library and create the Places object. After that the user can instantiate Agents and map them to the Places collection using the createAgents() function with the following signature:

```
template<typename AgentType, typename AgentStateType>
   Agents* createAgents(int handle, void *argument, int argSize, int nAgents,
      int placesHandle, int maxAgents =0, int* placeIdxs =NULL);
```

Parameter "*handle*" is a unique numerical identifier for the collection of Agents, "*argument*" and "*argSize*" define the arguments that can be passed to the Agent object at creation, "*nAgents*" is the number of agents to be initially created, "*placesHandle*" identifies the grid of places over which the new agents collections will be instantiated. Parameters "*maxAgents*" and "*placeIdxs*" are optional, and are assigned the default values by the library in case a user provides no values. "*maxAgents*" is the maximum number of agents that will be spawned in the system during the course of the computation. The default value for this parameter is *nAgents**2. And "*placeIdxs*" is the array of Place object indexes of size *nAgents*, which defines the location of instantiation for all the agents.

As part of initialization, Agents are mapped to the Places grid using the "*placeIdxs*" array, provided by the user, or the default map, which we implemented as part of this work. The default map randomly distributes Agents over a grid of Places. An array of agents' locations is created on CPU, using C++ standard library random number generator, and then copied to the GPU, where it's used in the instantiation of Agent objects. By creating the array on CPU, we can avoid the use of the cuRAND library, which can be quite inefficient. The algorithm we use to generate Place indexes for the Agents is as follows:

```
void getRandomPlaceIdxs(int idxs[], int nPlaces, int nAgents) {
   int curAllocated = 0;

   // If there is more than 1 agent per place, allocate agents evenly over space:
   if (nAgents > nPlaces) {
      for (int i=0; i<nPlaces; i++) {
         for (int j=0; j<nAgents/nPlaces; j++) {
                idxs[curAllocated] = i;
                curAllocated ++;
            }
        }
    }

    // Allocate the remaining agents randomly:
    std::unordered_set<int> occupied_places;
    while (curAllocated < nAgents) {
        unsigned int randPlace = rand() % nPlaces; //random number from 0 to nPlaces
        if (occupied_places.count(randPlace)==0) {
            occupied_places.insert(randPlace);
            idxs[curAllocated] = randPlace;
            curAllocated++;
        }
    }
}
```

When creating Agents, the library allocates more GPU memory space for the agent collection than the initial number of agents specified. The system allocates the "*maxAgents*" number of objects (can be defined by a user) but keeps the extra agents in an inactive state. Those inactive objects are then used when a user wants to create new agents through replication. Allocating all of the memory for agents once at the beginning of the simulation allows avoiding frequent dynamic memory allocation throughout the simulation, which can be a severe bottleneck on the GPU.

Following the architecture established by Harts work [2], we are splitting the behavior and state of Agents into two different classes, where the behavior class, Agent, contains all the functions as well as a single pointer to a generic AgentState class that contains all the fields. An array of concrete behavior classes are instantiated both on the host(CPU) and device(GPU), as well as an array of state classes of equal size.

Each Agent object, instantiated on GPU, holds a pointer to the Place object, instantiated on GPU as well, where it resides, and each Place object holds an array of Agents, which reside in that place.

When a user subsequently wants to see some Agents' properties on CPU (e.g., print the agents' locations), only the AgentState array is copied from GPU to CPU.

4.2. Agent Migration

One of the challenges of implementing dynamic agents support is designing the agent migration mechanism, in particular, the algorithm of resolving conflicts when several agents try to migrate to the same destination. In case of MASS CUDA, we also wanted to make it possible for the user of the library to control the conflict resolution rules specific to their application. Because all the agent threads run in parallel, we also need to manage a data race between several threads trying to modify the same Place object simultaneously.

There are several solutions to the problem of managing agent migration in a thread-safe manner:

a) **Let Agent threads write into the Place object without specific conflict resolution rules.** In case several agents try to migrate to the same destination, each following thread will overwrite the entry of the previous agent. So, the last agent thread to access the Place object gets that place. The benefit of this conflict resolution scheme is the fast speed of execution because there is no thread synchronization and thus no related overhead. However, the disadvantage of this algorithm is the non-deterministic nature of the simulation, because the result depends on the order of threads getting to a certain execution point.

b) **Manage data races between Agent threads through atomic functions.** When several Agent threads try to migrate to the same Place, the *atomicCAS()* function from the CUDA standard library is used to select an agent with the smallest index. Using this algorithm makes the simulation reproducible. However, it does not allow users to modify the conflict resolution algorithm between migrating agents. One of the requirements of the MASS CUDA library is to hide all CUDA implementation details, so we cannot expect users to manipulate CUDA atomic functions.

c) **Save all the Agents, which try to migrate to a Place, into an array and perform conflict resolution in a separate function.** This is the algorithm that we developed as part of this thesis work. Following this approach, each place has an array of agents that want to migrate to this place. Agents register their intent by saving their pointer into that array. For example, if in a system agents can only migrate one cell North, West, South or East, the location of all possible migrating agents will look as pictured in Figure 4.1.A and the array of potential migrating agents stored in that place will look as pictured in Figure 4.1.B. After all agents registered their intent to migrate, each Place performs conflict resolution based on a certain algorithm (can be provided by a user) and selects agents to accept. Selected agents then migrate to the places that accepted them as part of *manageAll()* function. The benefit of this approach is that a user can specify the conflict resolution rules. This approach also has drawbacks; in particular, the range of agent migration should be specified in the library parameters as it defines the size of the incoming agents' array in each Place. However, limited migration range is a reasonable assumption in many spatial ABMs.

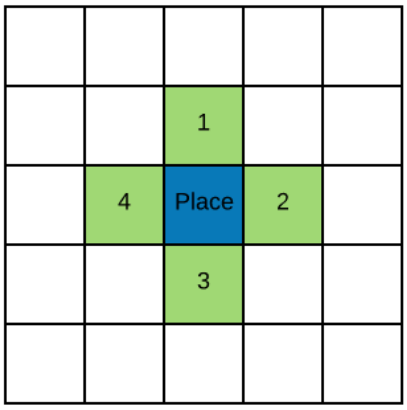

Figure 4.1.A: Potential migrating agents' locations.

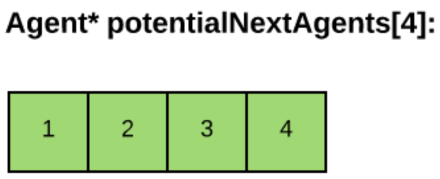

Figure 4.1.B: Array of potential migrating agents in a Place object.

As you can see in Figure 4.2, the speed of execution of different migration conflict resolution algorithms is different. The fastest is the library without specific conflict resolution rules (option (a) above), the second fastest is the library with the default conflict resolution using atomic functions (option (b)) and the slowest is our custom algorithm using an array of incoming agents in each Place (option (c)). Even though the option (c) is 40% slower than option (a), we chose it to be the conflict resolution mechanism for the primary version of MASS CUDA, as it is the most versatile and supports user-defined conflict resolution rules. If users do not require custom conflict resolution rules, they can choose to use the version of the library without deterministic conflict resolution or the version with atomic functions. So, MASS CUDA users have a choice between more functionality through overloadable conflict resolution function or faster execution.

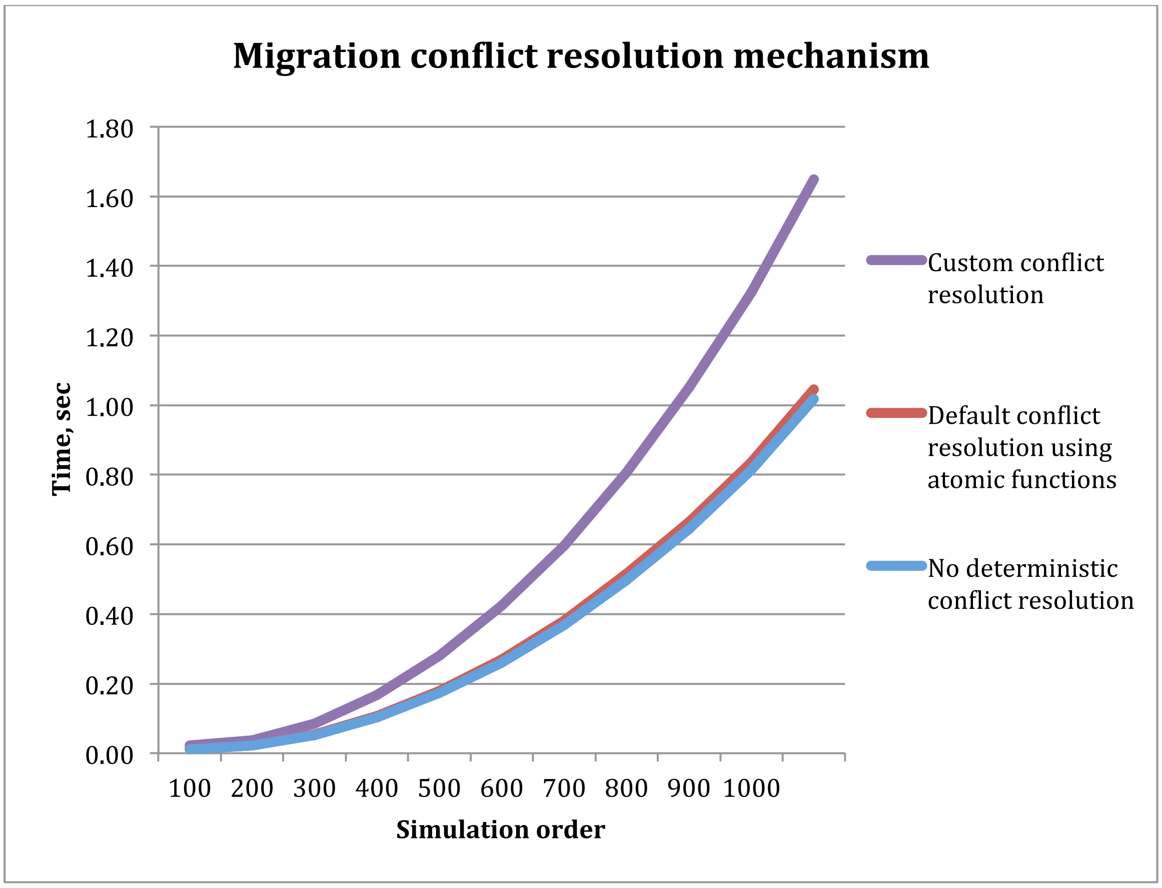

Figure 4.2: Performance of the SugarScape simulation depending on the MASS CUDA library migration conflict resolution mechanism

4.3. Agent Replication

Another challenge we faced while implementing fully functional dynamic agents, as part of MASS CUDA, was agent replication due to the difficult memory management. Because in CUDA we cannot allocate memory for the new agents from within the GPU kernels, the sufficient amount of memory should be allocated beforehand.

The way we addressed this challenge in MASS CUDA is by allocating extra space holding inactive objects at Agent collection instantiation time. Then, when a user asks to replicate a new agent, we pick the next available Agent object, set its members to the proper values and activate the object. GPU data model holds a counter to the current furthest allocated Agent object, and the counter is incremented using *atomicAdd()* function from the CUDA standard library. In the current work, agent replication is not evaluated as part of the large-scale performance measurement experiments, as our performance-benchmarking app (SugarScape) does not require the creation of new agents. We check the functional correctness of the implemented agent replication mechanism using a test program.

4.4. Agent Termination

Another technical decision we had to make while implementing dynamic agents was the memory management algorithm during agent termination. When the user calls *terminateAgent()* function on the Agent object, we do not deallocate the respective memory, as it can be time-consuming and will partition the memory space allocated for agents. Instead, we deactivate that object and ignore it for further computation.

Due to limitations in research scope, we did not implement garbage collection as part of MASS CUDA. Garbage collection for terminated objects can be implemented in multiple ways. One approach that can be taken involves sorting the agent array based on whether the agent is alive or dead [11]. The part of the array with dead agents can then be used for spawning new agents during replication stage. Another way to reuse the memory occupied by the terminated agents is to use the stochastic parallel allocation strategy [12] at the agent replication stage. This algorithm tries to match each gravid (about to reproduce) cell to a unique empty cell by letting gravid agents look a random offset to the right for a dead agent in parallel. Implementation and performance evaluation of different garbage collection algorithms as part of MASS CUDA was left outside of the scope of the current research but is a good future research topic.

To sum up, we implemented the support for dynamic agents as part of MASS CUDA, in particular, we proposed and implemented the functions of agent instantiation and mapping to space, agent migration, agent replication, and agent termination. As part of agent migration implementation, we discussed three different approaches for migration conflict resolution: no deterministic migration conflict resolution, migration conflict resolution using CUDA atomic functions and our custom algorithm for migration conflict resolution involving maintaining an array of all the Agents trying to migrate to a Place as part of that Place. We described the advantages of each of the algorithms regarding execution speed and functionality.

5. Performance Analysis

To measure the effectiveness of the techniques we implemented for spatial simulation performance optimization and the relative performance of the dynamic agent system as part of MASS CUDA, we run a series of experiments with simulations of different sizes. We also compared the results with the execution time of the sequential CPU-based versions of these simulations.

5.1. Evaluation Environment

The resources required for the successful implementation, testing and performance measurement of the current thesis requires specific hardware and software: modern CPU running Linux OS with CUDA-compatible GPU and the NVIDIA® CUDA® Toolkit installed.

The equipment we used to collect data about applications' performance has the specifications shown in Table 5.1.

Processor:	Intel Xeon CPU E5-2630 v3
Processor Clock Speed:	2.40GHz
Operating System:	Ubuntu Linux 4.4.0
RAM:	32 GB
Compiler:	CUDA 8.0 toolkit (NVCC)
Graphics Card:	GeForce GTX Titan
CUDA Compute Capability:	3.5
CUDA Cores (Streaming Multiprocessors):	2688
GPU Max Clock rate:	876 MHz
Memory Clock:	6.0 Gbps
Memory Bandwidth:	288.4 GB/sec

Table 5.1: Specifications of the hardware used in performance evaluation experiments

5.2. Performance Analysis of a Static System (Heat 2D)

To analyze the performance of the Spatial Simulation we used the Heat2D application, which is a simulation of heat transfer in a mass following the Euler method.

The simulation was run using a square space of values, and the sizes provided are the length of a single side, so a simulation of order 100 actually has the simulation space of size 100 by 100 and a total of 10,000 Place elements. Each simulation was run at the various sizes for 3000 iterations with heat being applied for the first 2700 iterations. Results were not displayed throughout the simulation in order to isolate run time from other, unrelated I/O performance factors.

Aside from the implementation of Heat2D using MASS CUDA version described in this thesis work, we also run experiments with the previous MASS CUDA library version, described in the work by Hart [2], the sequential CPU implementation of the application and the direct GPU implementation, which uses plain CUDA calls.

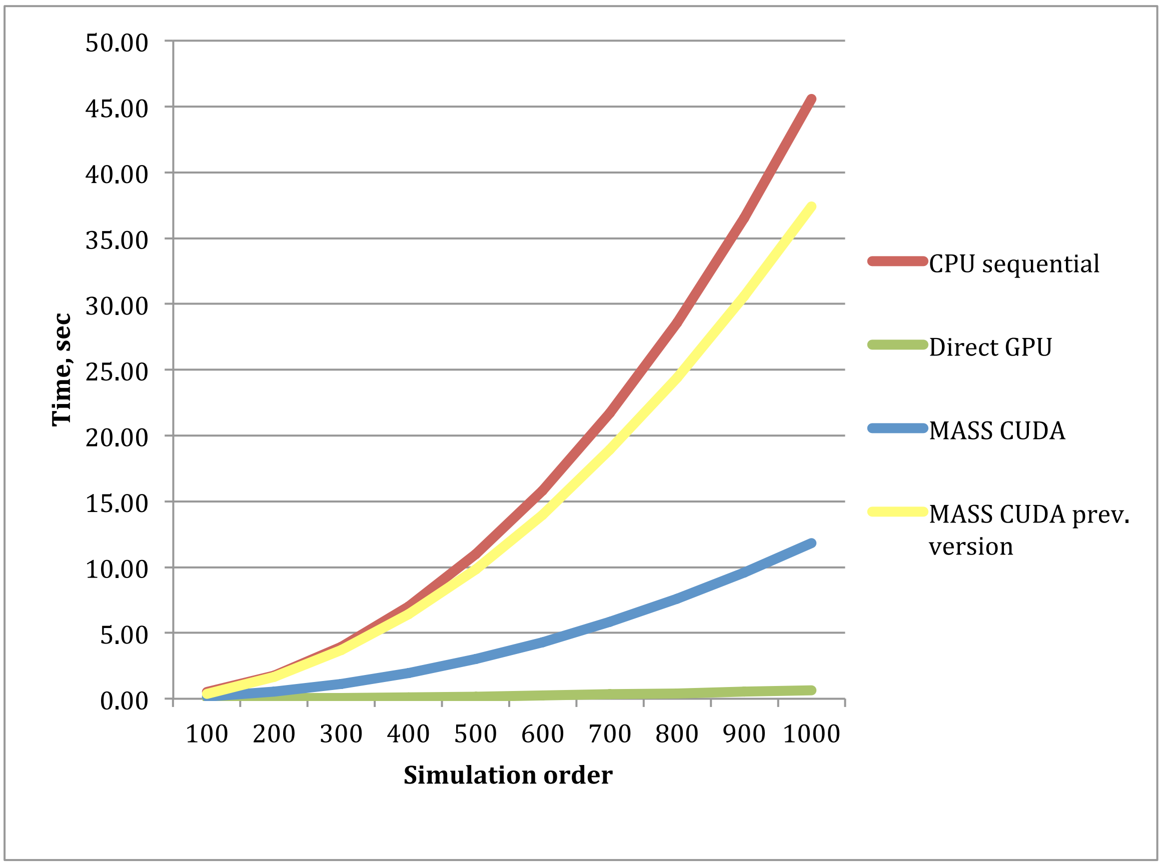

Figure 5.1: Execution time of different implementations of the Heat2D application

As can be observed from Figure 5.1, the current version of MASS CUDA library provides a significant speed-up for the chosen spatial simulation compared to its predecessor. It shows a 3.2 times improvement compared to the previous version of the library, which brings us up to a 3.9 times speed-up compared to the sequential CPU implementation of 1000 by 1000 simulation. The MASS CUDA app still takes more time than the direct GPU implementation of Heat2D using bare CUDA calls, which is an expected result, as library adds the overhead of additional layer of abstraction on top of CUDA code.

So, the experiments show that the techniques we implemented for optimizing the spatial simulation as part of MASS CUDA - the use of constant memory to store data frequently accessed by threads in different blocks; combining exchangeAll() and callAll() functions to execute in one GPU kernel call to avoid context switch; and selecting the optimal launch configuration of kernels –provided a significant speed-up compared both to sequential CPU version of the simulation and to the previous version of the library.

5.3. Performance Analysis of a Dynamic System (SugarScape)

For the purpose of assessing the performance on the agent support in MASS CUDA, we chose SugarScape application, because it's a widely known simulation problem and because it requires both spatial and agent functionality of the library. The application simulates the behavior of ant colony in the presence of sugar (nutrition source). Ants metabolize sugar and produce pollution, migrate to available places with more sugar and die if no more sugar is available around them.

Similar to Heat2D, SugarScape experiments are run using a square simulation space of varying sizes. Each simulation was run for 100 iterations and results were not displayed throughout the simulation in order to isolate run time from unrelated performance factors.

For the benchmarking purposes, we also developed versions of the SugarScape running on the CPU in a sequential manner and direct GPU implementation, executing bare CUDA kernel calls. For this performance comparison we do not have data from the previous MASS CUDA library version as it did not implement support of dynamic agents and thus it would not be possible to implement SugarScape on top of it.

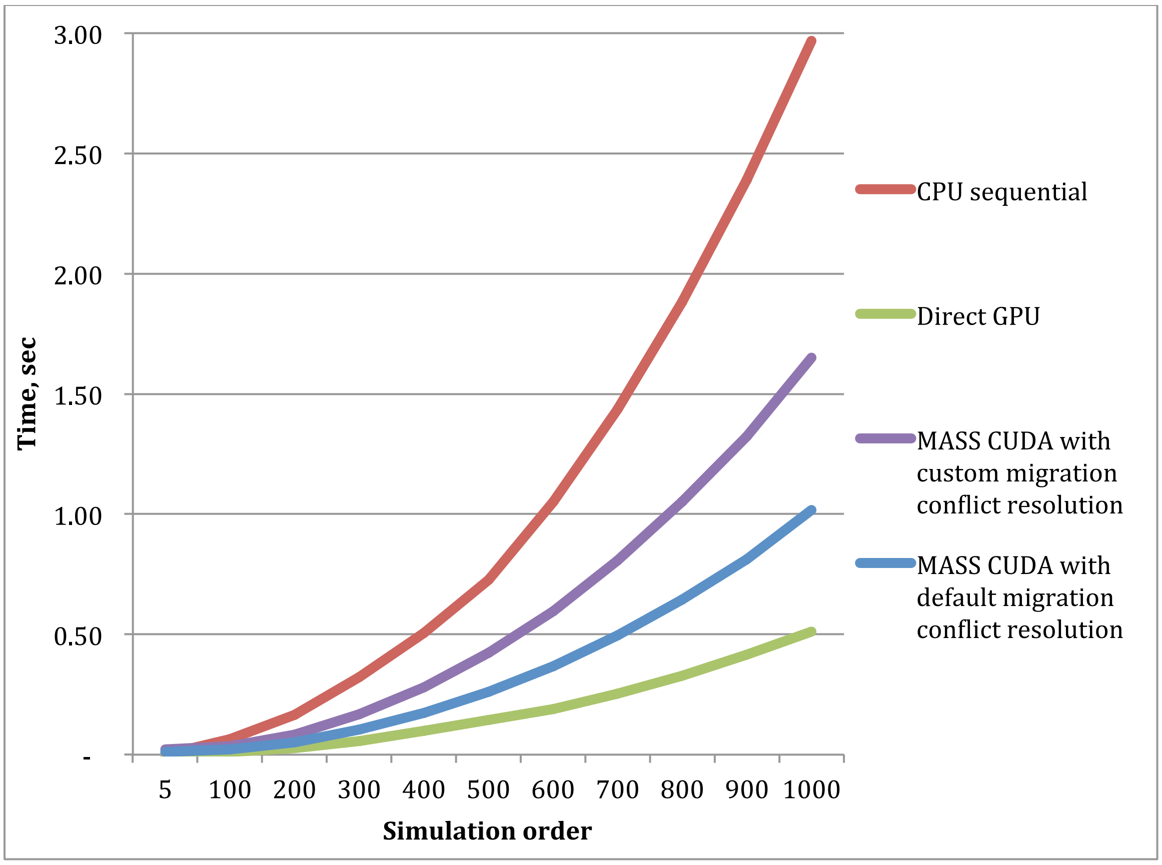

Figure 5.2: Execution time of different implementations of the SugarScape application

As you can see from Figure 5.2, the new version of MASS CUDA with custom migration conflict resolution provides a 1.8 times speed-up compared to the sequential implementation. If the application does not require custom conflict resolution rules, then MASS CUDA can provide up to 2.9 times speed up for the bigger simulation sizes using the default migration conflict resolution algorithm. Similarly to the Heat2D, we can observe that MASS CUDA does add some overheat on top of bare GPU implementation of the same problem, but it also offers significant improvement in programmability and can accommodate users without the knowledge of CUDA language or GPU architecture.

To achieve good simulation performance of the dynamic agent functionality, we considered GPU architecture and limitations when selecting implementation techniques and algorithms. For example, during agent instantiation and mapping we used random number allocation on CPU instead of the slow cuRAND library on GPU. And for agent termination and replication we used the agent pooling technique, which allocates a big chunk of memory and then keeps track of active agents in the system. This approach allows avoiding frequent dynamic memory allocation throughout the simulation, which can be a severe bottleneck on the GPU. Additionally, some carryover of the optimization techniques effect from spatial simulation to the dynamic agent-based system could have occurred.

In summary, both benchmarking applications – spatial Heat2D simulation and dynamic agent-based SugarScape application – demonstrate good performance speed up compared to the sequential CPU execution. Optimization techniques that resulted in increased spatial simulation efficiency are: (1) the use of constant memory to store data frequently accessed by threads in different blocks; (2) combining *exchangeAll()* and *callAll()* functions to execute in one GPU kernel call to avoid context switch; and (3) selecting the optimal launch configuration of kernels. The efficiency of the dynamic agent-based model is due to the selection of implementation techniques and algorithms that work well with CUDA architecture and limitations, such as random number allocation on CPU instead of GPU and the use of agent pooling technique to avoid frequent dynamic memory allocation on GPU during the course of a simulation.

6. Conclusion

This thesis research is focused on improving the performance of the spatial simulations and implementing support for dynamic agents as part of MASS CUDA library, in particular, such functions as agent instantiation and mapping to space, agent migration, agent replication, and agent termination.

6.1. Summary of This Work Contribution

As part of the thesis, we describe three main techniques, which allowed us to achieve 3.9 times speed-up in execution time compared to the sequential CPU implementation of a spatial simulation. The techniques are the use of constant memory to store data frequently accessed by threads in different blocks, avoiding context switch between GPU kernels by combining the data exchange and data processing kernels into one, and selecting the optimal launch configuration of kernels. We also describe techniques that did not provide any performance improvement for the spatial simulation, namely the use of the "pragma unroll" for loop unrolling in GPU kernels and replacement of getter and setter functions in the object classes with direct member access.

We also implemented the support for dynamic agents as part of MASS CUDA, and describe the specific techniques used for implementing agent instantiation and mapping to space, agent migration, agent replication, and agent termination. As part of agent migration implementation, we discussed three different approaches for migration conflict resolution: no deterministic migration conflict resolution, migration conflict resolution using CUDA atomic functions, and our custom algorithm for migration conflict resolution involving maintaining an array of all the Agents trying to migrate to a Place as part of that Place. We described the advantages of each of the algorithms in terms of execution speed and functionality.

6.2. Next Steps

The two main areas of future work include the implementation of agent garbage collection as part of MASS CUDA and evaluating the extensibility of the library to multi-cluster systems. Implementation and performance evaluation of different agent garbage collection algorithms as part of MASS CUDA will increase the range of potential applications that can run on top of the library, including applications with highly dynamic agent spawning and termination. Evaluating the extensibility of the library functionality and implemented optimization techniques to multi-GPU clusters is important as it could result in even higher performance of ABM simulations.

Bibliography

1. T. Chuang, M. Fukuda, “A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems”, In Proc. of the 16th IEEE International Conference on Computational Science and Engineering - CSE 2013, pp. 143-150, Sydney, Australia, December, 2013. [Online]. Available: http://faculty.washington.edu/mfukuda/papers/cse2013_mass.pdf
2. Hart, Nathaniel B. “MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks”. Diss. University of Washington, 2015. Available: https://onedrive.live.com/view.aspx?resid=AFB10F7D10CB103C!36244&ithint=file%2cpdf&app=WordPdf&authkey=!APN9OEQ_n3ufVPk
3. Aaby, Brandon G., Kalyan S. Perumalla, and Sudip K. Seal. “Efficient Simulation of Agent-Based Models on Multi-Gpu and Multi-Core Clusters.” *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. 29.
4. Alberts, Samuel et al. “Data-Parallel Techniques for Simulating a Mega-Scale Agent-Based Model of Systemic Inflammatory Response Syndrome on Graphics Processing Units.” *Simulation* 88.8 (2012): 895–907.
5. Bleiweiss, Avi. “Multi Agent Navigation on the GPU.” *Games Developpement Conference*. N.p., 2009. 39–42.
6. Caggianese, Giuseppe, and Ugo Erra. “Exploiting Gpus for Multi-Agent Path Planning on Grid Maps.” *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 2012. 482–488.
7. Caggianese, Giuseppe, and Ugo Erra. “Gpu Accelerated Multi-Agent Path Planning Based on Grid Space Decomposition.” *Procedia Computer Science* 9 (2012): 1847–1856.

8. Campeotto, Federico, Agostino Dovier, and Enrico Pontelli. “Protein Structure Prediction on GPU: A Declarative Approach in a Multi-Agent Framework.” *Parallel Processing (Icpp), 2013 42nd International Conference on*. IEEE, 2013. 474–479.
9. Cecilia, José M. et al. “Enhancing Data Parallelism for Ant Colony Optimization on GPUs.” *Journal of Parallel and Distributed Computing* 73.1 (2013): 42–51.
10. Chen, W. et al. “Agent Based Modeling of Blood Coagulation System: Implementation Using a GPU Based High Speed Framework.” *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. N.p., 2011. 145–148.
11. D’Souza, Roshan M. et al. “Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis on Graphics Processing Units.” *Proceedings of the 2009 Spring Simulation Multiconference*. Society for Computer Simulation International, 2009. 21.
12. D’Souza, Roshan M., Mikola Lysenko, and Keyvan Rahmani. “SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates.” *Proceedings of Agent2007 Conference. Chicago, IL*. N.p., 2007.
13. Erra, Ugo et al. “An Efficient GPU Implementation for Large Scale Individual-Based Simulation of Collective Behavior.” *High Performance Computational Systems Biology, 2009. HIBI’09. International Workshop on*. IEEE, 2009. 51–58.
14. Hermellin, Emmanuel, and Fabien Michel. “Gpu Delegation: Toward a Generic Approach for Developping Mabs Using Gpu Programming.” *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2016. 1249–1258.
15. Hermellin, Emmanuel, and Fabien Michel. “Overview of Case Studies on Adapting MABS Models to GPU Programming.” *International Conference on Practical Applications of Agents and Multi-Agent Systems*. Springer, 2016. 125–136.

16. Hidayat, R. et al. “Multi-Agent System with Multiple Group Modelling for Bird Flocking on GPU.” *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. N.p., 2016. 680–685.

17. Husselmann, Alwyn V., and Ken A. Hawick. “Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with Gpus.” *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*. N.p., 2011. 100–107.

18. Kumar, Jitendra, Lotika Singh, and Sandeep Paul. “GPU Based Parallel Cooperative Particle Swarm Optimization Using C-CUDA: A Case Study.” *Fuzzy Systems (FUZZ), 2013 IEEE International Conference on*. IEEE, 2013. 1–8.

19. Laville, Guillaume et al. “MCMAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation.” *Euro-Par 2013: Parallel Processing Workshops*. Springer, Berlin, Heidelberg, 2013. 544–554.

20. Li, D. et al. “A Efficient Algorithm for Molecular Dynamics Simulation on Hybrid CPU-GPU Computing Platforms.” *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*. N.p., 2016. 1357–1363.

21. Li, Xiaosong, 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 (2016): 3313–3332.

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

23. NVIDIA. NVIDIA CUDA programming guide. Technical Report, 2013. (Available from: http://docs.nvidia.com/cuda/cuda-c-programming-guide)
24. Passos, E. et al. “Supermassive Crowd Simulation on Gpu Based on Emergent Behavior.” *Proceedings of the Seventh Brazilian Symposium on Computer Games and Digital Entertainment*. Citeseer, 2008. 70–75.
25. Passos, Erick Baptista et al. “A Bidimensional Data Structure and Spatial Optimization for Supermassive Crowd Simulation on GPU.” *Computers in Entertainment (CIE)* 7.4 (2009): 60.
26. Richmond, Paul et al. “High Performance Cellular Level Agent-Based Simulation with FLAME for the GPU.” *Briefings in bioinformatics* 11.3 (2010): 334–347.
27. Richmond, Paul, Simon Coakley, and Daniela Romano. “Cellular Level Agent Based Modelling on the Graphics Processing Unit.” *High Performance Computational Systems Biology, 2009. HIBI’09. International Workshop on*. IEEE, 2009. 43–50.
28. Richmond, Paul, Simon Coakley, and Daniela M. Romano. “A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA.” *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2009. 1125–1126.
29. Richmond, Paul, and Daniela Romano. “Agent Based Gpu, a Real-Time 3d Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the Gpu.” *Proceedings International Workshop on Supervisualisation*. N.p., 2008.
30. Sano, Yoshihito, Yoshiaki Kadono, and Naoki Fukuta. “A Performance Optimization Support Framework for Gpu-Based Traffic Simulations with Negotiating Agents.” *Recent Advances in Agent-Based Complex Automated Negotiation*. Springer, 2016. 141–156.

31. Shen, Zhen, Kai Wang, and Fenghua Zhu. "Agent-Based Traffic Simulation and Traffic Signal Timing Optimization with GPU." *Intelligent Transportation Systems (Itsc), 2011 14th International Ieee Conference on*. IEEE, 2011. 145–150.
32. Storti, Duane, and Mete Yurtoglu. CUDA for Engineers: An Introduction to High-Performance Parallel Computing. Addison-Wesley Professional, 2015.
33. Strippgen, David, and Kai Nagel. "Multi-Agent Traffic Simulation with CUDA." *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*. IEEE, 2009. 106–114.
34. Strippgen, David, and Kai Nagel. "Using Common Graphics Hardware for Multi-Agent Traffic Simulation with CUDA." *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. 62.
35. Torchelsen, Rafael P. et al. "Real-Time Multi-Agent Path Planning on Arbitrary Surfaces." *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 2010. 47–54.
36. Vigueras, Guillermo, Juan M. Orduna, and Miguel Lozano. "A GPU-Based Multi-Agent System for Real-Time Simulations." *Advances in Practical Applications of Agents and Multiagent Systems*. Springer, 2010. 15–24.
37. Wang, Kai, and Zhen Shen. "A GPU Based Trafficparallel Simulation Module of Artificial Transportation Systems." *Service Operations and Logistics, and Informatics (SOLI), 2012 IEEE International Conference on*. IEEE, 2012. 160–165.