
Article:
Chimeh, M.K. and Richmond, P. (2018) Simulating heterogeneous behaviours in complex systems on GPUs. Simulation Modelling Practice and Theory, 83. pp. 3-17. ISSN: 1569-190X

https://doi.org/10.1016/j.simpat.2018.02.002

Simulation Modelling Practice and Theory 83 (2018) 3–17

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Simulating heterogeneous behaviours in complex systems on GPUs

Mozhgan K. Chimeh*, Paul Richmond

University of Sheffield, United Kingdom

ARTICLE INFO

Article history:
Available online 10 February 2018

Keywords:
Agent Based Modeling
GPGPU
Data parallel algorithms
Simulation
FLAME GPU

ABSTRACT

Agent Based Modelling (ABM) is an approach for modelling dynamic systems and studying complex and emergent behaviour. ABMs have been widely applied in diverse disciplines including biology, economics, and social sciences. The scalability of ABM simulations is typically limited due to the computationally expensive nature of simulating a large number of individuals. As such, large scale ABM simulations are excellent candidates to apply parallel computing approaches such as Graphics Processing Units (GPUs). In this paper, we present an extension to the FLAME GPU[1] [1] framework which addresses the divergence problem, i.e. the challenge of executing the behaviour of non-homogeneous individuals on vectorised GPU processors. We do this by describing a modelling methodology which exposes inherent parallelism within the model which is exploited by novel additions to the software permitting higher levels of concurrent simulation execution. Moreover, we demonstrate how this extension can be applied to realistic cellular level tissue model by benchmarking the model to demonstrate a measured speedup of over 4x.

1. Introduction

A complex system is a system which involves a large number of interacting entities/individual agents. Agent Based Modelling (ABM) is a method of understanding the behaviours of such system. It allows the behaviour of entities comprising a complex system, to be described as a set of individual behaviours or rules. In order to explore emergent properties resulting from such systems, the behaviours and interactions between entities can be simulated.

When simulating complex models using ABMs, increasing the size of the model or behavioural complexity of the individuals leads to an increase in the computational requirements. The use of parallel computing resources offers a viable solution to constrain these requirements and sustain reasonable simulation times even for very large or complex systems. Modern Graphics Processing Units (GPUs) contain hundreds of arithmetic processing units that can be utilised to achieve significant acceleration for computationally intensive scientific applications. GPUs allow a personal computer to be transformed into a personal supercomputer, providing up to 12 Trillion Floating Point Operations per Second (TFLOPS) in consumer hardware (NVIDIA TITAN Xp). Whilst computationally powerful, GPUs differ significantly in the hardware design of modern CPUs.

* Corresponding author.
E-mail addresses: m.kabiri-chimeh@sheffield.ac.uk (M.K. Chimeh), p.richmond@sheffield.com (P. Richmond).
URL: http://mkchimeh.staff.shef.ac.uk (M.K. Chimeh), http://paulrichmond.shef.ac.uk (P. Richmond)

[1] Flexible Large-scale Agent-based Modelling simulation platform on GPU

https://doi.org/10.1016/j.simpat.2018.02.002

Writing programs which are able to utilise the high levels of parallel performance requires considerable knowledge of data parallel algorithm design and extensive optimisation skills.

At first glances, ABMs appear well suited to a GPUs architecture. Each of the individuals (or agents) within the system can be simulated in parallel. Unfortunately, this is not the case, one has to consider methods of handling communication (and hence synchronisation) among individual agents, sparsity within populations as a result of agent creation and death (life and death), and dealing with heterogeneous behaviours within the group (divergence). Solving these challenges requires algorithms to be designed or applied to each of this issues which uses data-parallelism to scale to the GPUs hundreds of tightly coupled vector processing units. FLAME GPU is an extended version of the FLAME (Flexible Large-scale Agent-Based Modelling Environment) framework and is a mature and stable Agent-Based Modelling simulation platform that enables modellers from various disciplines like economics, biology and social sciences to easily write agent-based models, specially for GPUs. Developed since 2008, FLAME GPU has previously addressed the challenges of communication among agents and life and death of agents [2]. The software is able to provide agent based modellers with the ability to target readily available GPUs capable of simulating many millions of interacting agents. Previous work has demonstrated that performance can easily exceed that of traditional CPU based simulators [3]. One of the key design considerations is the use of a high level ABM syntax to abstract the complexities of the underlying GPU architecture away from modellers. This ensures that modellers can concentrate on writing models without the need to acquire specialist knowledge typically required to program GPU architectures.

Aside from relatively simple (or artificial benchmark) models, complex systems are typically heterogeneous with respect to the exhibited behaviour of individuals. In other words, agents within ABM rarely have identical sets of rules over their complete life-cycle. For instance, an agent representing a typical biological cell might have a number of distinct behavioural stages as it differentiates between cell types. The issue of divergence or divergent behaviour within populations of agents is largely unsolved by FLAME GPU. The FLAME GPU user guide actively encourages the use of large population sizes to ensure that the GPU device (which requires many parallel operations on agents) can be fully utilised when executing threads corresponding to agent behaviour. Typically populations of many thousands of agents are required to ensure good utilisation and as such agent behaviours often contain high levels of divergence.

The paper outlines best practices and considerations for model developers to systematically address agent heterogeneity on GPU architectures. The objective of this paper is therefore to propose a specific set of design considerations for implementation of agent models to ensure efficient execution on the GPU by reducing divergence. To evaluate the performance improvement, the design considerations have been tested with novel implementations (Section 5). The results show considerable improvements on the state-of-the-art, demonstrating the first formal provision of guidelines for best practice in addressing agent heterogeneity on GPU architectures in a consistent manner.

This paper aims to consider the problem of agent divergence offering the following novel contributions;

- We present an extension to FLAME GPU which exploits inherent parallelism within models to demonstrate efficient concurrent execution of divergent behaviours.
- We introduced key design considerations for modellers which encourage high levels of parallelism within a model whilst minimising the effect of divergent behaviour.
- We apply this extension to a heterogeneous cellular level biological model of tissue wound formation to demonstrate a simulation performance speedup of 4x on model sizes of up to 131k agents.

The rest of this paper is organised as follow: Section 3 describes the related issues and challenges of multi-agent simulation on GPUs. Section 4 presents design considerations required to implementing a heterogeneous model with high degrees of model parallelism. Section 5, reports the result of our experimental evaluation. Finally, we draw our conclusions in Section 7.

2. Background

Simulation is a tool for researchers to study and better understand the behaviour of a system, as well as predicting its performance. A number of different methods may be used to represent the system based on the characteristic of the model. Typically complex system can be represented as top-down by using sets of equations to model system level behaviour or bottom up by modelling the individuals with the system as agents. Simulation of the ABM, often referred to as Multi-Agent Simulation (MAS), is a method of studying complex systems [4] which provides a natural modelling approach as often the individual levels behaviour are well understood.

Due to the availability of increasing levels of computing power, a number of simulation frameworks have been developed and applied in different fields of engineering and science. Examples include Swarm [5], NetLogo [6], FLAME [7], Mason [8], and MCMAS [9]. D-MASON (Distributed Mason) is the distributed version of MASON MAS simulation framework developed to address the limitations on scalability (increasing the number of population) with respect to distributed hardware configurations. It uses remote agent invocation to communicate between agents to effectively address issues of synchronisation between many communicating CPU processors. FLAME uses a formal method of agent representation based on communicating X-Machines which solves synchronisation of communication by ensuring that only indirect communication through messages is permitted. The use of message boards ensures that communication can be limited to only agents where there

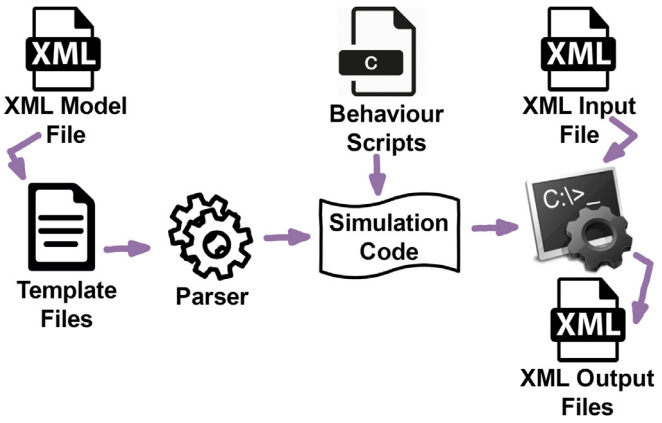

Fig. 1. FLAME GPU code generation process.

are specific dependencies on information. With respect to any distributed framework, there is an inherent latency in communication over a network which may lead to poor performance [10]. This is particularly true of simulations with high levels of communication compared to the level of computation. As such distributed methods are well suited to simulations with low numbers of highly complex agents which infrequently communicate, for example, simulation of large economic systems [11].

As an alternative to a distributed model, GPUs afford large levels of parallelism within a shared memory system and can, therefore, avoid communication latencies evident in distributed approaches. GPUs have been widely applied in many scientific research domains to accelerate applications and achieve significant computational performance improvements [12]. With respect to MAS, there exists a number of domain specific simulation frameworks for traffic [13], soil [14] and blood coagulation system [15]. The work by D'Souza [16] describes a methodology for implementing discrete space (cellular) ABMs on GPU. However, the work does not consider divergence as a result of focusing only on homogeneous (sugarscape) ecology models. FLAME GPU [17] is a generalised framework that permits modellers from any domain to write a model without an understanding of the GPU architecture. Fig. 1 shows FLAME GPU code generation process which automatically translates a high level model description to optimised GPU code described in a series of code generation templates.

The major design issue with FLAME GPU is that when applied to model systems of complex heterogeneous behaviours, the code generations creates a simulation which demonstrates divergent execution paths leading to poor computational performance. Divergent execution paths are a result of nearby threads requiring different execution paths at conditional branches during execution with Single Instruction Multiple Data (SIMD) processors. Control flow divergent problems on GPU have been studied more generally outside of the MAS domain in previous works [18–21]. However, to the best of authors knowledge, the divergence problem for the specific use cases unique to ABM execution have never been addressed (Fig. 2).

3. Challenges of simulating ABMs on the GPU

Graphics Processing Units (GPUs) contain thousands of lightweight cores with high bandwidth access to its dedicated on-chip memory. Compute Unified Device Architecture (CUDA) is a C/C++ based API built around a heterogeneous programming model for NVIDIA GPUs. The programming model exposes the GPU device to developers as a grid of thread blocks executing data parallel programs (kernels). Kernel configuration is managed by the host (CPU) with kernels executed on the device (GPU) [22].

On GPUs, thousands of threads execute simultaneously, however, threads operate on a different set of data (data parallelism). Prior to the execution of GPU instructions, a fixed number of threads are grouped together in small SIMD groups called warps. Threads within a warp have consecutive IDs/indices and the warp size is equal to the SIMD width of the GPU core (32 in all current architectures). The multiple warps within the user defined blocks of threads are selected by the GPUs zero-overhead hardware scheduler and switched to execute on a single multi-processing unit with concurrent execution of instructions within the SIMD processors (otherwise known as CUDA processors). It is vital that threads within a warp do not diverge as this results in serial evaluation of instructions (intra-warp divergence). Where conditional code with divergent paths (within a warp) must be evaluated both code paths must be evaluated with non participating threads masking execution of the instruction. This results in performance degradation, in the worst case 32 individual codes paths may need to be evaluated by every thread.

Other sources of divergence can occur when warps within a thread block finish their execution at a different times (i.e. inter-warp divergence) [24] or different memory latencies (e.g.: cache misses, uncoalesced accessed pattern) cause divergence in memory access time between warps.

In ABM different types of heterogeneous agents may exist within a model. When behaviour is expressed as a program (or script) for execution on the GPU, the complexity of the behavioural rules increases the heterogeneity within the population

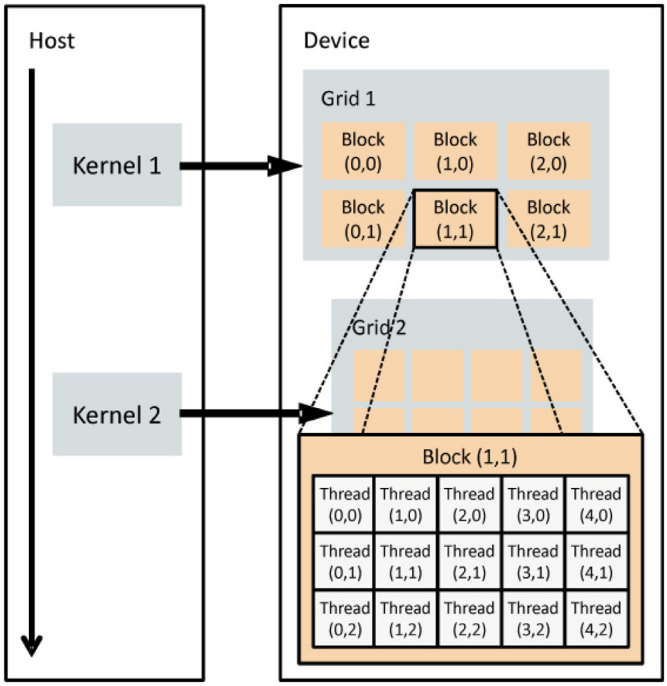

Fig. 2. CUDA programming model [23]. Each kernel is assigned to a grid, each containing a number of blocks.

leading to potentially divergent code paths. The resulting control flow divergence has significant impact on the simulation performance. There are various potential solutions for control flow divergence optimisation which require threads with similar execution paths to be grouped prior to execution. However, control flow divergence problem within agents cannot be entirely eliminated due to the underlying heterogeneous behaviours exhibited.

Other sources of divergence can potentially occur within MAS as a result of sparse population caused by agent's life and death. This form of divergence has broadly been addressed in FLAME GPU and other specific GPU MAS implementations. For example, the use of stream compaction with Thrust [25] to avoid sparse data and fix non contiguous data layouts in memory as a results of removal or creation of an agent, has been discussed in [1].

Within FLAME GPU there is no other source of control flow intra-warp divergence other than the conditional branch caused by the agent state. State-based agent representation is used to specifically avoid any divergence elsewhere in the model execution. Although there is a possibility of potential divergence issue in the building of data structures used for messaging, however, the building of these data structures should be considered in isolation and these are outside of the scope of improving agent execution efficiency.

3.1. State-based representation

FLAME GPU uses an underlying state based representation that separates agents into distinct states. The state machine based agent representation allows agents to be separated into state lists. Processing of agents within the state list ensures coalesced memory access. Any state transition or agent function requires the movement of agents from one state to another (a.k.a transferring agents to a new state list).

When transitioning agents between state lists, there are additional operations such as stream compaction which occur before and after the agent function execution. Similar to agent function, these operations are able to operate within the associated streams. The overhead cost associated with management of the state lists is a small percentage of the total agent function execution time. This has been demonstrated in a previous work [2] by breaking down the simulation time during a single simulation step.

When models are designed with large numbers of states, the stream compaction filtering can effectively be used to minimise behavioural divergence and ensure coalesced memory access during agent behaviours' execution. Whilst effective, heavy use of state based representation to minimise the divergence, leads to smaller (less divergent) state lists of agents which are unable to fully utilise the GPUs many cores. In order to ensure a good utilisation of resources and keep the GPUs many threads busy, large population sizes are required. Low population sizes within state lists result in smaller grid block configurations during the computation of the agent scripts. The impact of small grid sizes is that there are too few effective warps to either occupy the GPUs numerous multi-processors or too few warps per multi-processor to hide the latency of expensive memory operations. For example, the minimum agent count required to fully utilise an NVIDIA TITAN X

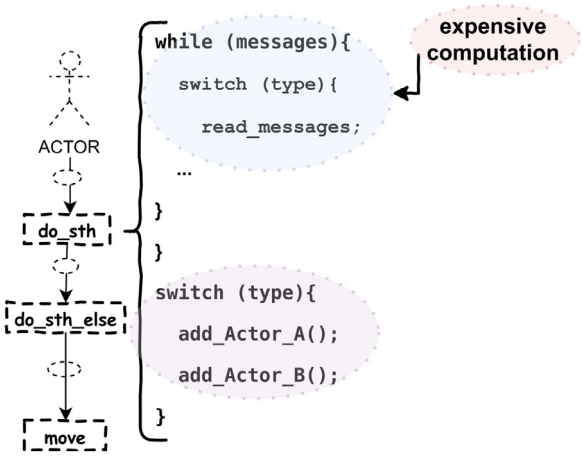

Fig. 3. Divergence in agent behaviour function can be minimised by transforming the model using multiple agent types or states. During this transformation, the computationally expensive operations are evaluated by model designers and converted to a new agent function.

(Pascal) GPU implementing the FLAME GPU Boids flocking model [26] can be calculated by considering the ratio of potential active warps on an single multi-processors to the maximum number of active warps supported by the multiprocessor (the occupancy) for the highest occupancy agent function (in this case the input_locations function). The occupancy in this case dictates the maximum number of supported thread blocks of a given block size (chosen to maximise the occupancy metric) required to fully utilise a single multi-processor. The 28 multi-processors are each able to support 2 blocks of 1024 threads requiring a minimum population size of 57,344. Future hardware will require even larger population sizes as the trend in GPU computing design is for increasing device nodes.

The impact of under-utilisation is significant and as such divergent behaviour is often overlooked in model design as a mechanism to ensure good device utilisation. Overlooking divergence in this way potentially inhibits the abstraction of the GPU from modellers. As a result of designing a model to have large population numbers, there is a requirement to understand why this will have an impact on the performance of the model once executed.

In the case of FLAME GPU, the issue of divergence can be resolved by expressing models with high use of states to first minimise divergence (at the expense of utilisation) and then concurrently executing independent agent functions on different multi-processors through the use of CUDA streams. Each stream is a queue of GPU operations, either memory movements or code execution, which can be independently dequeued where no dependencies exist. This effectively allows independent streams to run concurrently with respect to each other to obtain good overall device utilisation with execution of independent code on many small grid block configurations. We have implemented a change to the FLAME GPU code generation templates which enables the use of CUDA streams. This modification therefore enables simultaneous execution of independent agent functions ultimately exploiting parallelisms within a model, expressed through the use independent state lists or agent types. The implications are that reducing divergence through state groupings, no longer reduces the overall performance for small state list population sizes.

3.2. Design considerations

In order for models to take advantage of CUDA streams within FLAME GPU, divergent paths within agent function scripts should be minimised, particularly where large amounts of divergent code or control flow is used. This rearrangement in agent specification requires a modeller to create a model using design considerations that maximises the use of states and therefore minimises divergence within behaviour scripts.

Fig. 3 shows a simple example FLAMEGPU model where the agent behaviour function contains conditional statements. In order to minimise the divergence within the FLAMEGPU model, the computationally expensive operations (e.g: messaging) within the divergent code are extracted and are transformed into new functions representing a new model with both a *multiple agent type* or *multi state model* design. In both model designs the aim is to increase independent functions which can be simultaneously executed by removing complex conditional code from the assigned agent functions.

- *Multi-state model* is a model where an agent has multiple states. In a multi state model, the divergence within an agent function is handled through agent function conditions where execution of the function is controlled by the value of the agent variable. The condition determines if the function should be applied to all or none of the agents within a specific state (See pseudo code in Fig. 4(b) which corresponds to diagram in Fig. 5(b)).
- *Multi-agent model* is a model with multiple agent types. In a multi agent type model, the divergence is handled by creating new model with multiple agent types manually derived from divergent paths in agent functions. Agent behaviours are defined separately for each agent type (See pseudo code in Fig. 4(c) which corresponds to diagram in Fig. 5(c)).

```
/*ACTOR agent function*/
do_sth(agent_ACTOR *agent){
    if (agent->type == A){
        ...
    }
    else if (agent->type == B){
        ...
    }
}
```

(a) A model with divergence

```
/*ACTOR agent function*/
do_sth_A(agent_ACTOR *agent){
    ...
}
/*ACTOR agent function*/
do_sth_B(agent_ACTOR *agent){
    ...
}
```

(b) Transformation to a single agent model with multiple states

```
/*ACTOR_A agent function*/
do_sth_A(agent_ACTOR_A *agent){
    ...
}
/*ACTOR_B agent function*/
do_sth_else_B(agent_ACTOR_B *agent){
    ...
}
```

(c) Transformation to a multi agent model

Fig. 4. Minimising the divergence in the model (a) by expressing the model using two alternative representation of the agents (b,c).

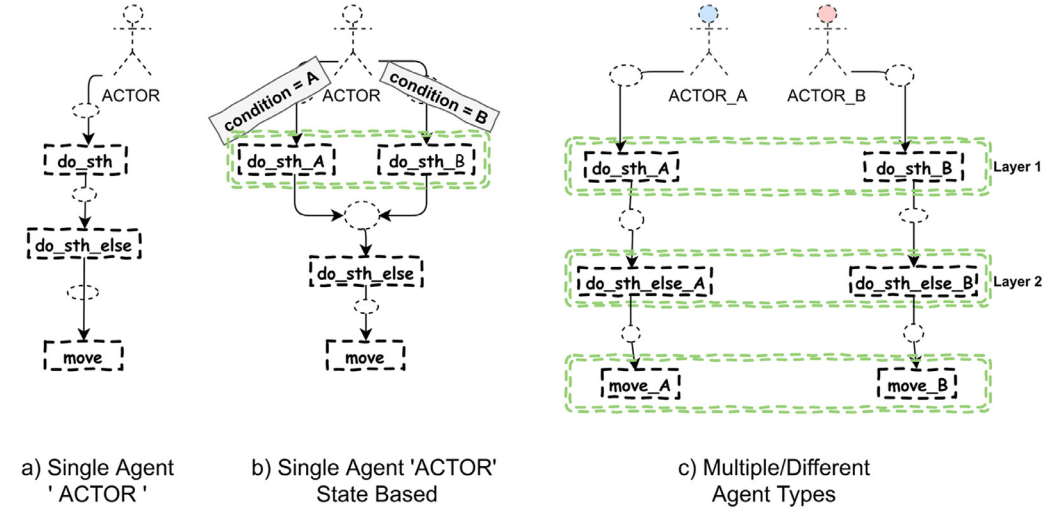

Fig. 5. Minimising Divergence by converting a single agent with multiple divergent functions (left) into either a single agent with multiple states (centre) or multiple agents with a single state (right). Green boxes indicate inherent parallelism within the model which can be exploited using concurrent function execution.

Increasing the number of independent functions allows functions to then run simultaneously through CUDA streams eliminating the branches within the agent behaviour code only the non-compute intensive operations will run sequentially.

Note that the functionality of both models (multi state and multi agent type) are equivalent.

Given a highly divergent FLAMEGPU model characterised by agent functions with conditional statements or switches on a particular agent variable, we transform a single agent model to a *multi type agent* or a *multi state model*. as follows:

1. Examine the code to find branches (e.g: conditional statements).
2. Within each divergent code path inside the agent behaviour function, extract expensive computations (e.g: reading messages) along with any statements outside of the conditional statement which use variables updated inside the divergent code.

```
/*ACTOR agent function*/
do_sth(agent_ACTOR *agent){
    if (agent->type == A){
        agent->x++;
        ...
    }
    else if (agent->type == B){
        agent->y++;
    }
    agent->x++;
    agent->y++;
}
```

(a) With variable dependency

```
/*ACTOR agent function*/
do_sth(agent_ACTOR *agent){
    if (agent->type == A){
        agent->x++;
        ...
    }
    agent->y++;
}
```

(b) Without variable dependency

Fig. 6. Examples of variable dependency at different points within the agent function.

3. Create new agent functions from the extracted expensive behaviours.
4. If there is no variable dependency between the conditional statement and the code outside of it, choose the first representation of the agents (multi agent type model) (See Fig. 6(b)), otherwise, the choice would be to transform the model to a multi-state agent (See Fig. 6(a)):
   (a) Define function conditions within each newly created agent function, so that the function should only be applied to agents which meet the condition and in the correct state (Fig. 4(b))
   (b) Transform the single agent model to multi agent type by creating an agent per "type" where each agent performs behaviour derived from the single divergent code path (Fig. 4(c))

   Note that the newly created functions also include the statements with variable dependency (will be explained further later in this section).
5. Operations remaining in the agent behaviour are defined as a separate agent function for each agent type and will be serialised. Note that there is no variable dependency between these operations with the divergent code.

To automate such procedure, a complex parsing and dependency analysis technique would be required. For example, agent behaviour should be described in a way to allow the extrapolation of divergent statements and presently this cannot be done. This is user's responsibility to re-write the model and this can only be addressed with human intervention.

Fig. 5 shows the two user interventions which can be undertaken to express the model using an alternative representation of the agents. The result of these user interventions (following proposed design considerations) produces an alternative flowchart representation.

Fig. 5(a) shows a single ACTOR agent with multiple agent functions. Agent functions in this model will execute sequentially and suffer from divergence as a result of the conditional statements within agent functions. The divergence can be eliminated through two choices depicted in Fig. 5(b) and (c).

As previously explained, this choice depends on the number of variables shared between the code outside of conditional statement and inside it. An already defined variable `agent->x` in an agent function is said to be *live* at a the given point if there is a control flow from within the conditional statement to a use of `agent->x`. This means that the result of an computation inside of the conditional statement is used in the code outside of the condition. Pseudo codes in Fig. 6 show examples of variable dependency where variables are updated outside of the conditionals. In Fig. 6(a), variables `agent->x` and `agent->y` live outside of the conditional statement and are being used in the code outside of the conditional statements whereas the agent function in Fig. 6(b) does not have any variable dependency.

Fig. 5(b), shows a single ACTOR agent model which handles divergence through a state based representation. Depending on the ACTOR agent variable type, (i.e.: with value of A or B), agents will have divergent behaviours. Another proposed low divergent solution to the first model, would be to divide the divergent paths associated to a specific agent ACTOR type into agent types or states (Fig. 5(c)). More specifically, the model can be specified with multiple agent types, where each agent performs behaviour derived from the single divergent code path. Fig. 5(c), shows the newly created model that has two agents of `ACTOR_A` and `ACTOR_B` and the `do_sth` agent function is divided to two kernel functions of `do_sth_A` and `do_sth_B`.

To summarise, if there is variable dependency between the conditional statement and the code outside of it, then the divergence within the agent function can be reduced by using the choice illustrated in Fig. 5(b). If the variables updated in the divergent paths were not used in the outside code within the agent function, then newly formed functions can be created based on these paths as shown in Fig. 5(c). Note that the second approach is preferable when there is no variable dependency as there will be no dependency or function conditions throughout the code.

Note that the pseudo codes in Fig. 4 correspond to the structured flowcharts in Fig. 5. Furthermore, pseudo codes in Fig. 6 show variable dependencies within the example `do_sth` agent function for the example model shown in Fig. 5(a). Given the rules, for this particular example model in Fig. 5(a), the transformation in Fig. 5(b) is preferable if the agent function was Fig. 6(a), otherwise, the model should be expressed using the representation of the agents in Fig. 5(c).

The advantage of re-arranging the model (using either multiple states within one agent or different agents type for each `ACTOR`) is that the simulation has less divergent code, more parallelism within the model. Additionally reduction of divergence may also lead to a reduction in communication where previously divergent paths conditionally examined message inputs.

Note that having two different states within an agent is no different to two different agents as FLAME GPU ultimately stores all agent data in separate state lists. Without the use of CUDA streams a model expressed with inherent parallelism will result in sequential execution of all agent functions. A key design consideration is proposed for FLAME GPU models which if followed ensures that the abstraction of GPU can be maintained;

Divergent behaviour should be minimised within the specification of agent behaviour in favour of the use of agent state lists or separate agents.

Modellers should express models in this way to promote inherent parallelism which can then be exploited by the use of CUDA streams. In next section, we use a case study to implement a low divergence model using the second (separate agent) method.

4. Case study of implementing a low divergence model

The Keratinocyte (cell) model [27] was developed as part of the Epithelium project [28] to model the in-vitro behaviour of various types of skin epithelial (tissue) cells. It is included within the FLAME GPU SDK as one of the example models. The model is used to study the interactions among cells during normal tissue growth and to understand source of dysregulation in pathology (e.g: compromised wound healing, malignant development).

The model includes various behaviours such as cell cycles, differentiation, apoptosis, and migration. Cell cycling includes both cell growth and division leading to the addition of agents to the model. Differentiation behaviour represent changes to cells type from keratinocyte stem cells capable of infinite divisions to fully differentiated superficial cells that cannot divide and are ultimately lost from the tissue surface. Apoptosis simulates the removal of dead agents from the simulation cycle and during migration, cells actively move within a 3D continuous space environment.

Within the FLAME GPU Keratinocyte model, cells are represented by spheres which are required to form tight bonds within cell colonies. Due to the force based movement of agents within the model there is a possibility of undesirable physical overlaps among cells, especially following cell divisions. Cell overlaps are resolved through the iterative application of repulsive force applied to cells the same way as any inter cellular forces that would lead to cell growth or motility. Resolution of the overlapping behaviours is achieved when the population reaches a convergent stable state, measured by a global condition representing the minimal acceptable overlap. To achieve this behaviour, a number of agent functions are defined specifically to perform the force resolution. Alternatively a fixed number of force resolution iterations can be applied to approximate the steady state. This force resolution behaviour has previously been implemented and its performance was compared to the CPU version [29]. In comparison with single threaded implementation within the original FLAME framework the model exhibits multiple orders of magnitude in performance improvement.

4.1. Divergence problem in tissue wound model

The Keratinocyte model has a number of different cell types represented by a single integer agent variable. Within each iteration of the simulation agents (cells) output their location and cell type (where 0 = stem cell; 1 = transit amplifying (`TA`) cell; 2 = committed cell (`COMM`); 3 = corneocyte (`CORN`); 4 = `HACAT`) to the message lists for other cells to read. Throughout the simulation, each cell performs specific behavioural rules depending on its type. The value of the type variable is determined based on a cells position in cell cycle and based on the differentiation behaviour may change throughout the cells life-cycle.

The model was initially designed this way to ensure that the GPU would be well utilised by guaranteeing a large agent population size. The agent behaviours exhibit significant code divergence with respect to behaviour which is compounded by unnecessary message reading. It was designed this way to achieve good utilisation of the hardware as a divergent free model would have had population sizes which were too low. As discussed previously, conditional statements leads to divergence and ultimately degrade performance.

5. Results and discussion

To reduce the impact of divergence within the Keratinocyte model, we have re-specified it following the design considerations concluded in the previous section (Section 3). This ensures more parallelism within the model and hence concurrent execution of agent functions through CUDA streams.

Automating this approach is akin to auto parallelism. Despite years of research, fully automatic parallelisation has proven intractable in general. Although, it would be ideal to provide an algorithm that would automatically reduce divergent com-

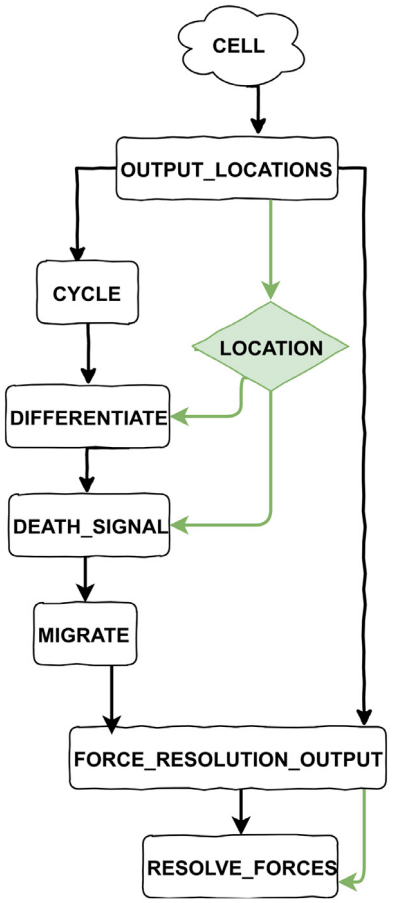

Fig. 7. Keratinocyte model state diagram with branch divergence.

putation for a given arbitrary model, automation of such task is exceptionally difficult. New methods of agent specification behaviour would be required to infer where there are divergent statements at a fine grained level (below that of agent functions). Currently, there is no clearly defined method of describing agent behaviour in this way.

Fig. 7 shows the original state diagram obtained from the XML model specification file for the Keratinocye model with higher code divergence and Fig. 8 is the proposed model with less divergent code. The new model has multiple agent types, meaning there is a different agent type for each cell type and there is a distinction between messages for specific agent (cell) types. All agent functions have been rewritten to avoid any conditional dependencies. The modified version of the Keratinocyte model consists of multiple agents, four message types, a single initialisation function and multiple agent functions which include multiple birth and death allocations (Figs. 7 and 8). Note that fundamentally the models have equivalent behaviour. The equivalence has been validated statistically as both models are Monte Carlo based and the separation at behaviour into new agent types changes the initial random seed values.

In order to evaluate the performance of the proposed low divergence model and asses the impact of our proposed design consideration of FLAME GPU modification we have benchmarked the two models at varying population sizes. Experiments were conducted using NVIDIA Pascal-based GPUs to evaluate our technique. More specifically, all the experiments are performed on a single PC with an Intel i7-2600 quad core hyper-threaded processor (3.40 GHz), 16GB RAM and an NVIDIA TITAN X (Pascal) GPU with 3584 CUDA cores and 12GB of memory. The generated CUDA programs by FLAME GPU are compiled using NVIDIAs CUDA 8.0 compiler, `nvcc`.

Each simulation was performed for 1000 iterations (at which point the population reaches a stable state) and the population was scaled up from 512 to 131k. The same initial states were used for both models and in the initial configuration, cells are randomly distributed at a constant density. A fixed number of 10 force iterations was used to resolve cell contact forces. Fig. 9 shows the population size with respect to the iteration number. The curves in plot show that for various initial population number, the population growth rate during the simulation is the same. Fig. 9 therefore demonstrates that for any initial population number, when the density is constant, the model shows the same behaviour after a fixed simulation cycles. Fig. 10, visualises the Keratinocyte model at iteration 0, 500, and 1000 with initial population of 512 randomly distributed at a constant density.

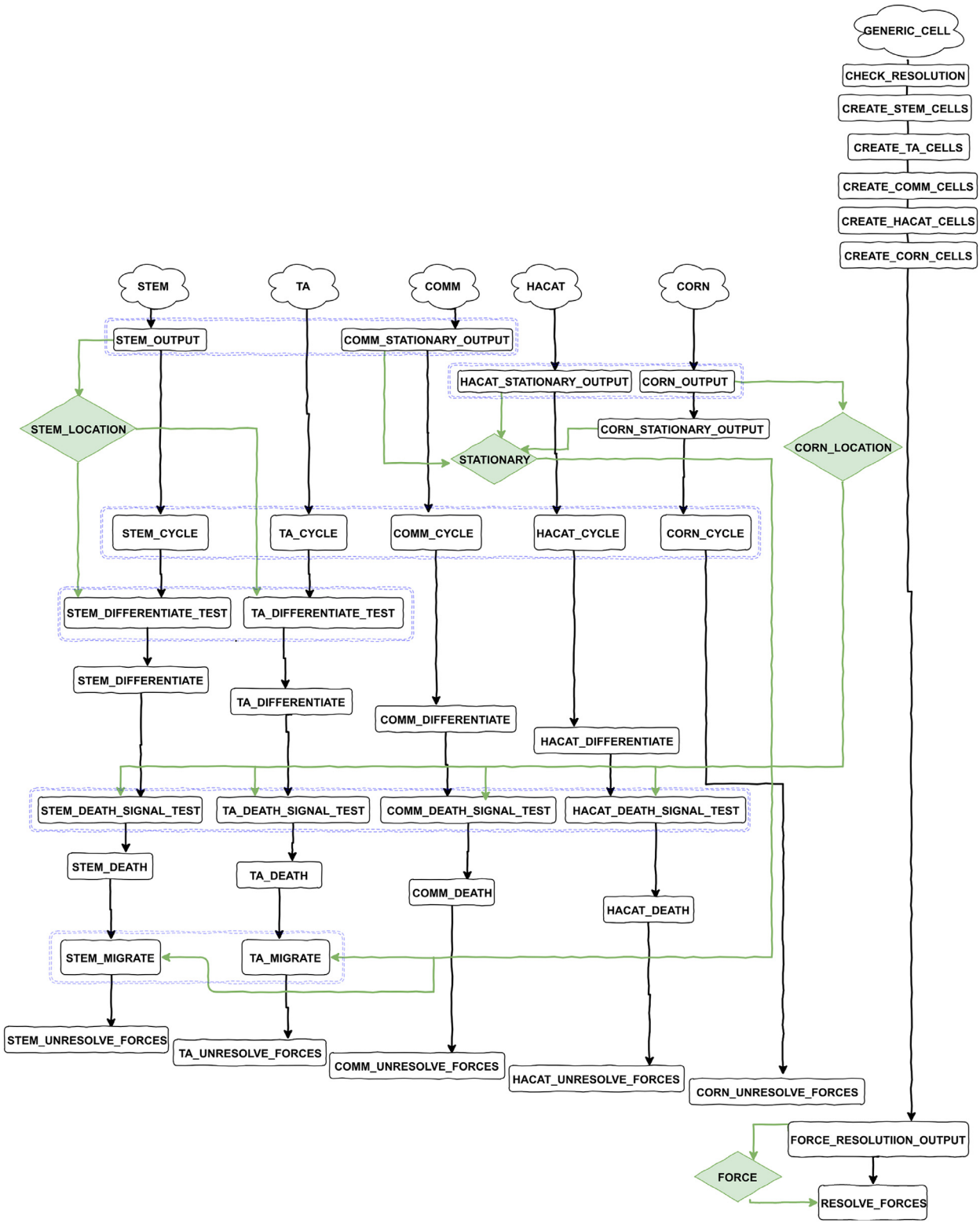

Fig. 8. Keratinocyte model state diagram without branch divergence. Blue dotted boxes represent inherent parallelism within the model. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

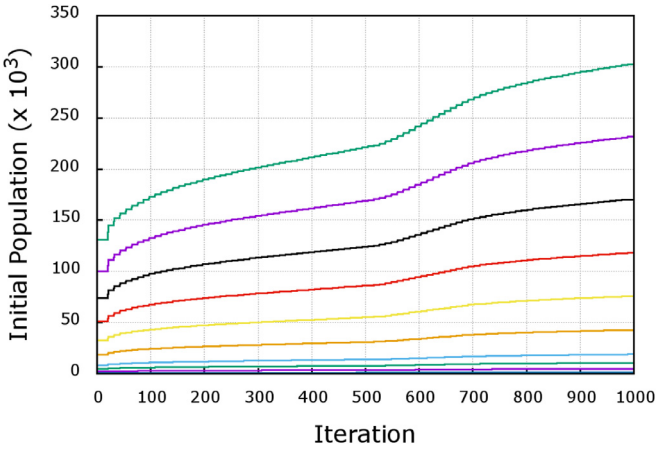

Fig. 9. Population growth rate with respect to the iteration number. Graph is equivalent for both high and low divergent model implementations.

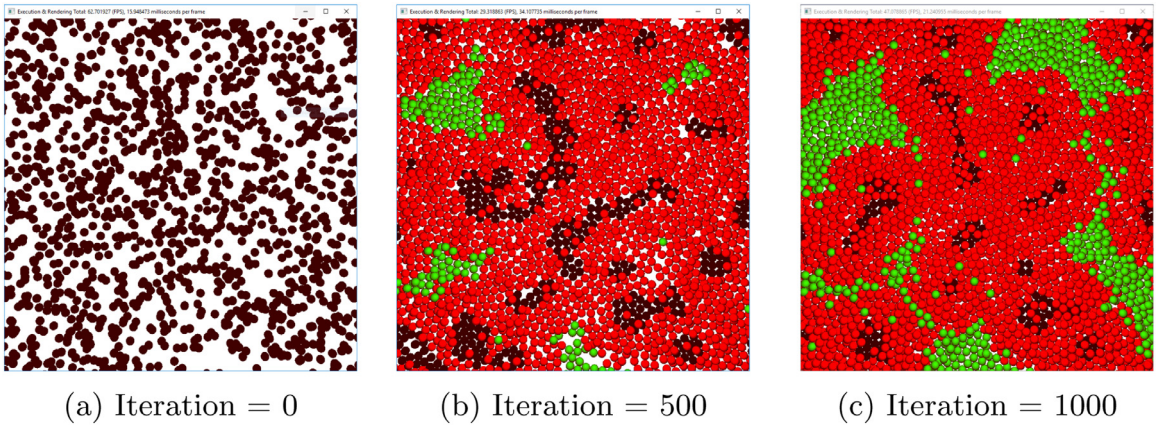

Fig. 10. Visualization of Keratinocyte model at iteration 0, 500, 1000. Cells are rendered as spheres.

5.1. Overall performance

Fig. 11 demonstrates the performance comparison of total run-time as well as cell behaviour functions[2] excluding the force resolution functions which contain no divergence in either case) for the two Keratinocyte models. The modified low divergent model achieved an overall run-time speedup of up to 4x comparing to the model with branch divergence. The plot shows for smaller number of population, the modified model takes slightly longer due to overhead in configuring the concurrent streams with the increased number of agent functions. The performance of the cell behaviour functions improve the cell behaviour aspect of the simulation performance by 32x due to divergence elimination in the modified model.

In addition to the overall performance of the model, we used other metrics to demonstrate the efficiency of our proposed control flow divergence method. In Section 5.2, we examine the standard agent function execution overhead to the negligibility of the overhead except for specific cases such as new agent creation. Moreover, in Section 5.3, we explain how the device is kept oversubscribed in the modified model to reduce the impact of workload imbalance and achieve significant performance improvement. Furthermore, a more detailed analysis on the impact of divergence elimination in the modified model is done through measuring warp execution efficiency. The experimental results is discussed in Section 5.4.

5.2. Agent function overhead

To measure the overhead of standard agent function execution, we measured the time required to execute the kernel only over agent function (that also includes the kernel). Results in the Table 1 reflect the fact that standard agent function overhead is less than 1%. The table shows overhead percentage for of the original model running for 1000 iterations with an initial population of 131k. The high levels of overhead seen for the `cycle` agent function is due to the creation

[2] Cell cycles, differentiation, apoptosis/death, and migration.

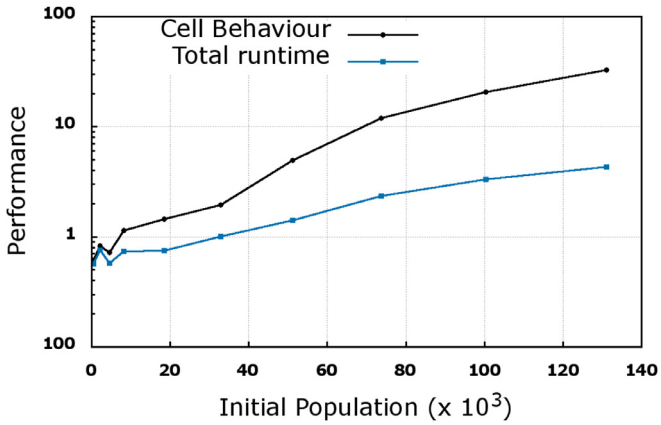

Fig. 11. Relative Speedup of the Keratinocyte model (logarithmic scale) for total run-time and cell behaviour functions (cell cycles, differentiation, apoptosis/death, and migration).

Table 1
Breakdown of simulation time for 1000 simulation steps of the original Keratinocyte model with an initial population of 131k.

Agent Function	Total Runtime (s)	Kernel only Runtime (s)	Overhead (%)
migrate	48.00	48.09	0.18
death_signal	47.88	47.96	0.17
differentiate	48.37	48.43	0.13
cycle	0.01	0.10	84.47
output_location	0.018	0.79	97.67
resolve_forces	9.03	9.56	5.57
force_resolution_output	0.18	15.07	98.75

of new agents, while the overhead of the two `output_location` and `force_resolution_output` is as a result of building data structure for efficient message parsing in subsequent layers in the same iteration [29]. The same observations can be made for the modified model, e.g. agent functions have negligible overhead except for those which require the building of data structures or generation of new agents. As explained in Section 3.1, the total overhead has a minor effect on overall performance.

5.3. Load imbalance

The CUDA Runtime system naturally maps blocks of threads executing a kernel to SMs. Load imbalance occurs when Streaming Multiprocessors (SMs) are not fully utilised or if multiple waves of thread blocks are not sufficient to keep the device busy. In either case some level of device under-utilisation may occur. The solution is to oversubscribe the device to keep the SMs occupied and sustain the performance for all but the final wave of thread blocks (which will always exist for non regular problem sizes).

Although state list size is smaller in the modified version of the Keratinocyte model (potentially leading to under utilisation), execution performed by threads on these lists occurs concurrently within the multiple streams to keep the SMs busy. With a large number of overall threads the device is oversubscribed and the impact of any imbalance is minimised. Without the use of CUDA streams the individual execution of FLAME GPU kernels would be serialised, effectively resulting in a large number of kernel executions with reduce device utilisation.

5.4. Warp efficiency

To observe the impact of divergence minimisation through the implementation of our design considerations, we profiled the warp execution efficiency (predicated and non-predicated averaged) of both original and modified model. Warp execution efficiency is the average percentage of active threads in each executed warp. On average, divergence elimination in the modified model with initial population of 131k, increased the warp execution efficiency of kernels from 58.29% to 82.19%. Higher warp execution efficiency indicates better performance as a result of effective utilisation of GPU compute resources.

6. Related work

The resource under-utilisation problem due to thread block level resource management has been observed in [30] and its limitation has been first identified and addressed in [31]. The resource fragmentation and under-utilisation due to branch divergence have been well studied [21,32–36]. Previous works on control flow divergence proposed various techniques to handle the issue at compiler/hardware level. In this section, we briefly describe some of these previous research works. Much existing research work focuses on hardware solutions such as warp scheduling to equalise execution time disparity for warps in the same thread block. An early work by Xiang et al. [24] uses a warp level resource management technique that dispatches a partial thread block to an SM, if it has resources for one or more warps. A partial thread block is launched when there are no resources to launch a full thread block. In this method, when a warp finished its execution, the allocated resource is released to be used by a new warp. One of the limitations to this approach is that the new thread block can be allocated if its shared memory requirement is met. This is due to the fact that the allocation and deallocation of the shared memory are done at the thread block level. Lee et al. proposes techniques to predict and identify critical warps at runtime [37,38]. While critical warps will be scheduled more frequently than others, the waiting time of warp for others is not eliminated in this approach. A Two-level warp scheduler to reduce the execution latencies was proposed by Narasiman et al. [39]. In the proposed method, warps are divided into groups and scheduled in round robin manner. Gebhart et al. [40] proposed a two-level warp scheduler focusing on energy efficiency for high throughput processors. There are exiting research work focusing on techniques to improve memory and cache to minimise the impact of long memory latencies [41–44].

Fung et al. [36] proposed a hardware implementation to dynamically regroup threads for more efficient Single Instruction Multiple Data (SIMD) branch execution on GPU. Hardware solutions require changes that may increase complexity and hardware cost in scheduling logic, register file, etc. In addition to hardware solutions, software-based techniques to minimise control flow (inter-warp) divergence have also been proposed. Software solutions can be applied to GPUs without any hardware changes. Chen and Shen [45] proposed a compiler transformation to reassign the tasks in sub-kernels to the parent threads. Anantpur and Govindarajan [46] proposed compiler based control flow linearisation technique to handle branch divergence by converting an unstructured control flow graph (CFG) to a structure CFG.

Other solutions were proposed to eliminate/minimise the branch and thread divergence [21,34,35,47].

In this work, the aim is to reduce intra-warp divergence of existing hardware and software models rather than use hardware solutions such as warp scheduling to hide warp execution latency. The above hardware approaches are orthogonal to our software-based approach and may stand to benefit anticipated cases of inter-warp divergence occurring naturally during runtime. Our approach of proposing design considerations to minimise intra-warp divergence is similar to that of the software based approaches however the FLAME GPU software uses the state lists grouping explicitly to avoid control flow divergence.

7. Conclusion

Simulation of complex systems through ABM provides a computational challenge due to the amount of computational performance required to simulate all individuals within a large population. The Graphics Processing Unit (GPU) presents a potential solution while creating other challenges related to the use of resources.

This paper has presented an overview on some of the challenges associated with simulating ABMs on the GPU, one of which is control flow divergence. We have proposed a design consideration and an improvement to the state of the art in GPU ABM software. The combination of these contributions eliminates thread divergence as a result of having conditional statements that would lead to irregularities in control flow and ultimately degrade simulation performance.

The proposed contributions were evaluated through the implementation of a Keratinocyte model. We compared the execution time of the transformed model with its original version on TITAN X GPU to demonstrate that our divergence optimisation solution reduced the overall impact of control flow divergence by 4x. With respect to the behavioural aspect of the model which specifically suffer from divergence we demonstrated a 32x speedup. This suggests that the non divergent, force resolution, part of the Keratinocyte model has a significant impact on overall performance. It is likely that the impact of our work will have considerably larger impact on models where divergence is more prevalent throughout the entire model design.

Acknowledgement

This work was supported by EPSRC fellowship “Accelerating Scientific Discovery with Accelerated Computing” (EP/N018869/1).

References

[1] P. Richmond, S. Coakley, D.M. Romano, A high performance agent based modelling framework on graphics card hardware with CUDA, in: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2, AAMAS '09, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2009, pp. 1125–1126. URL http://dl.acm.org/citation.cfm?id=1558109.1558172.

[2] P. Richmond, D. Romano, Template-driven agent-based modeling and simulation with CUDA, 2011,

[3] R. Chisholm, P. Richmond, S. Maddock, A Standardised Benchmark for Assessing the Performance of Fixed Radius Near Neighbours, Springer International Publishing, Cham, pp. 311–321. doi:10.1007/978-3-319-58943-5_25.
[4] P. Siebers, U. Aickelin, Introduction to multi-Agent simulation, CoRR (2008). arXiv:0803.3905. URL http://arxiv.org/abs/0803.3905.
[5] N. Minar, R. Burkhart, C. Langton, M. Askenazi, The swarm simulation system: a toolkit for building multi-agent simulations, - Working Paper 96-06-042 (1996).
[6] S. Tisue, U. Wilensky, NetLogo: Design and implementation of a multi-agent modeling environment, in: Proceedings of the Agent 2004 Conference on Social Dynamics: Interaction, Reflexivity and Emergence, Chicago, IL, 2004.
[7] M. Holcombe, S. Coakley, R. Smallwood, A general framework for agent-based modelling of complex systems, in: Proceedings of the 2006 European Conference on Complex Systems, 2006.
[8] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, G. Balan, MASON: a multiagent simulation environment, Simulation 81 (7) (2005) 517–527, doi:10.1177/0037549705058073.
[9] G. Laville, C. Lang, B. Herrmann, L. Philippe, K. Mazouzi, N. Marilleau, MCMAS: a toolkit for developing agent-based simulations on many-core architectures, Multiagent Grid Syst. 11 (1) (2015) 15–31, doi:10.3233/MGS-150227.
[10] A. Rousset, B. Herrmann, C. Lang, L. Philippe, A Survey on Parallel and Distributed Multi-Agent Systems, Springer International Publishing, Cham, pp. 371–382. doi:10.1007/978-3-319-14325-5_32.
[11] M. Holcombe, S. Coakley, M. Kiran, S. Chin, C. Greenough, D. Worth, S. Cincotti, M. Raberto, A. Teglio, C. Deissenberg, S. Van der Hoog, H. Dawid, S. Gemkow, P. Harting, M. Neugart, Large-scale modeling of economic systems, 2013.
[12] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008) 40–53, doi:10.1145/1365490.1365500.
[13] Z. Shen, K. Wang, F. Zhu, Agent-based traffic simulation and traffic signal timing optimization with GPU, in: 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC), 2011, pp. 145–150, doi:10.1109/ITSC.2011.6083080.
[14] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, L. Philippe, Using GPU for multi-agent soil simulation, in: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013, pp. 392–399, doi:10.1109/PDP.2013.63.
[15] W. Chen, K. Ward, Q. Li, V. Kecman, K. Najarian, N. Menke, Agent based modeling of blood coagulation system: implementation using a GPU based high speed framework, in: 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2011, pp. 145–148, doi:10.1109/IEMBS.2011.6089915.
[16] M. Lysenko, R.M. D'Souza, A framework for megascale agent based model simulations on Graphics Processing Units, J. Artif. Soc. Soc. Simul. 11 (4) (2008) 10. http://jasss.soc.surrey.ac.uk/11/4/10.html.
[17] D.P. Richmond, FLAME GPU technical report and user guide 1 (2012) 24. URL http://www.flamegpu.com/documentation.php.
[18] K.S. Perumalla, B.G. Aaby, Data parallel execution challenges and runtime performance of agent simulations on GPUs, in: Proceedings of the 2008 Spring Simulation Multiconference, SpringSim '08, Society for Computer Simulation International, San Diego, CA, USA, 2008, pp. 116–123. URL http://dl.acm.org/citation.cfm?id=1400549.1400564.
[19] Y. Liang, M.T. Satria, K. Rupnow, D. Chen, An accurate GPU performance model for effective control flow divergence optimization, Trans. Comp.-Aided Des. Integ. Cir. Sys. 35 (7) (2016) 1165–1178, doi:10.1109/TCAD.2015.2501303.
[20] I. Chakroun, M. Mezmaz, N. Melab, A. Bendjoudi, Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm, Concurrency and Computation: Practice and Experience (2012). URL https://hal.inria.fr/hal-00731859.
[21] T.D. Han, T.S. Abdelrahman, Reducing branch divergence in GPU programs, in: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, ACM, New York, NY, USA, 2011, pp. 3:1–3:8, doi:10.1145/1964179.1964184.
[22] Y. Cai, S. See, GPU Computing and Applications, Springer Publishing Company, Incorporated, 2014.
[23] R. Di Salvo, C. Pino, Image and video processing on CUDA: state of the art and future directions, in: Proceedings of the 13th WSEAS International Conference on Mathematical and Computational Methods in Science and Engineering, MACMESE'11, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2011, pp. 60–66. URL http://dl.acm.org/citation.cfm?id=2074857.2074871.
[24] P. Xiang, Y. Yang, H. Zhou, Warp-level divergence in GPUs: characterization, impact, and mitigation, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 284–295, doi:10.1109/HPCA.2014.6835939.
[25] J. Hoberock, N. Bell, Thrust: a parallel template library, 2010. URL http://www.meganewtons.com/.
[26] C.W. Reynolds, Flocks, herds and schools: a distributed behavioral model, SIGGRAPH Comput. Graph. 21 (4) (1987) 25–34, doi:10.1145/37402.37406.
[27] T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood, S. MacNeil, An integrated systems biology approach to understanding the rules of keratinocyte colony formation, J. R. Soc. Interface 4 (17) (2007) 1077–1092, doi:10.1098/rsif.2007.0227.
[28] D. Walker, J. Southgate, G. Hill, M. Holcombe, D. Hose, S. Wood, S.M. Neil, R. Smallwood, The epitheliome: agent-based modelling of the social behaviour of cells, BioSystems 76 (13) (2004) 89–100. Papers presented at the Fifth International Workshop on Information Processing in Cells and Tissues. http://doi.org/10.1016/j.biosystems.2004.05.025.
[29] P. Richmond, D. Walker, S. Coakley, D. Romano, High performance cellular level agent-based simulation with FLAME for the GPU, Briefings in Bioinformatics 11 (3) (2010) 334, doi:10.1093/bib/bbp073.
[30] D.B. Kirk, W.-m.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 2, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
[31] D. Tarjan, K. Skadron, On demand register allocation and deallocation for a multithreaded processor, 2011, US Patent App. 12/649,238 URL http://www.google.co.uk/patents/US20110161616.
[32] J. Meng, D. Tarjan, K. Skadron, Dynamic warp subdivision for integrated branch and memory divergence tolerance, SIGARCH Comput. Archit. News 38 (3) (2010) 235–246, doi:10.1145/1816038.1815992.
[33] J.D. Collins, D.M. Tullsen, H. Wang, Control flow optimization via dynamic reconvergence prediction, in: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 37, IEEE Computer Society, Washington, DC, USA, 2004, pp. 129–140, doi:10.1109/MICRO.2004.13.
[34] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, S. Yalamanchili, Simd re-convergence at thread frontiers, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, ACM, New York, NY, USA, 2011, pp. 477–488, doi:10.1145/2155620.2155676.
[35] M. Rhu, M. Erez, The dual-path execution model for efficient GPU control flow, in: Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), HPCA '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 591–602, doi:10.1109/HPCA.2013.6522352.
[36] W.W.L. Fung, I. Sham, G. Yuan, T.M. Aamodt, Dynamic warp formation and scheduling for efficient GPU control flow, in: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, IEEE Computer Society, Washington, DC, USA, 2007, pp. 407–420, doi:10.1109/MICRO.2007.12.
[37] S.-Y. Lee, C.-J. Wu, CAWS: criticality-aware warp scheduling for GPGPUworkloads, in: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14, ACM, New York, NY, USA, 2014, pp. 175–186, doi:10.1145/2628071.2628107.
[38] S.-Y. Lee, A. Arunkumar, C.-J. Wu, CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads, in: Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15, ACM, New York, NY, USA, 2015, pp. 515–527, doi:10.1145/2749469.2750418.
[39] V. Narasiman, M. Shebanow, C.J. Lee, R. Miftakhutdinov, O. Mutlu, Y.N. Patt, Improving GPU performance via large warps and two-level warp scheduling, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, ACM, New York, NY, USA, 2011, pp. 308–317, doi:10.1145/2155620.2155656.
[40] M. Gebhart, D.R. Johnson, D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, K. Skadron, Energy-efficient mechanisms for managing thread context in throughput processors, SIGARCH Comput. Archit. News 39 (3) (2011) 235–246, doi:10.1145/2024723.2000093.

[41] A. Jog, O. Kayiran, N.C. Nachiappan, A.K. Mishra, M.T. Kandemir, O. Mutlu, R. Iyer, C.R. Das, Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance, SIGPLAN Not. 48 (4) (2013) 395–406, doi:10.1145/2499368.2451158.
[42] O. Kayiran, A. Jog, M.T. Kandemir, C.R. Das, Neither more nor less: optimizing thread-level parallelism for GPGPUs, in: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 157–166.
[43] T.G. Rogers, M. O'Connor, T.M. Aamodt, Cache-conscious wavefront scheduling, in: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, IEEE Computer Society, Washington, DC, USA, 2012, pp. 72–83, doi:10.1109/MICRO.2012.16.
[44] T.G. Rogers, M. O'Connor, T.M. Aamodt, Divergence-aware warp scheduling, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, ACM, New York, NY, USA, 2013, pp. 99–110, doi:10.1145/2540708.2540718.
[45] G. Chen, X. Shen, Free launch: optimizing GPU dynamic kernel launches through thread reuse, in: Proceedings of the 48th International Symposium on Microarchitecture, MICRO-48, ACM, New York, NY, USA, 2015, pp. 407–419, doi:10.1145/2830772.2830818.
[46] J. Anantpur, R. Govindarajan, Taming Control Divergence in GPUs through Control Flow Linearization, Springer, Berlin, Heidelberg, pp. 133–153. doi:10.1007/978-3-642-54807-9_8.
[47] F. Khorasani, R. Gupta, L.N. Bhuyan, Efficient warp execution in presence of divergence with collaborative context collection, in: Proceedings of the 48th International Symposium on Microarchitecture, MICRO-48, ACM, New York, NY, USA, 2015, pp. 204–215, doi:10.1145/2830772.2830796.