**Proceedings Paper:**
Alzahrani, E., Simons, A.J.H. orcid.org/0000-0002-5925-7148 and Richmond, P. orcid.org/0000-0002-4657-5518 (2020) Data aware simulation of complex systems on GPUs. In: 2019 International Conference on High Performance Computing & Simulation (HPCS). 2019 International Conference on High Performance Computing & Simulation (HPCS), 15-19 Jul 2019, Dublin, Ireland. Institute of Electrical and Electronics Engineers (IEEE) , pp. 567-574. ISBN 9781728144856

https://doi.org/10.1109/hpcs48598.2019.9188054

2019 International Conference on High Performance Computing  Simulation

# Data Aware Simulation of Complex Systems on GPUs

1st Eidah Alzahrani*
*Department of Computer Science*
*University of Sheffield*
S1 4DP, Sheffield, UK
e.alzahrani@sheffield.ac.uk
**Al Baha University, AlBaha, Saudi Arabia*

2nd Anthony J H Simons
*Department of Computer Science*
*University of Sheffield*
S1 4DP, Sheffield, UK
a.j.simons@sheffield.ac.uk

3rd Paul Richmond
*Department of Computer Science*
*University of Sheffield*
S1 4DP, Sheffield, UK
p.richmond@sheffield.ac.uk

***Abstract*—GPUs have been demonstrated to be highly effective at improving the performance of Multi-Agent Systems (MAS). One of the major limitations of further performance improvements is in the memory bandwidth required to move agent data through the GPU's memory hierarchy. This paper presents a formal model for data aware simulation and an empirical study into the impact of minimising data movement on performance. This study proposes a method that can be applied to the simulation of complex systems on GPUs to extract required data from agent behaviour during simulation time and how this information can be used to reduce data movement. The FLAME GPU software has been extended to demonstrate this technique. Three benchmark experiments have been applied to evaluate the overall reduction in simulation execution time under specific criteria. The results of the comparison between the current and new system show that reducing data movement within a simulation improves overall performance with up to 4.8x speedup reported.**

***Index Terms*—Memory access reduction, FLAME GPU, Agent Based Modelling, Simulation**

## I. INTRODUCTION

Agent-Based Modelling (ABM) is a technique that is used to simulate the actions and reactions of individuals (as agents). Agents can communicate with each other and the environment based on simple rules, and this makes ABM a suitable approach for simulating complex systems. By using multi-core central processing units (CPUs), distributed systems and accelerators such as graphic processing units (GPUs), high performance computing (HPC) creates a suitable environment for handling complex operations. These operations may include, but are not limited to, big data, simulating complex systems, performing large-scale simulations, and dealing with similar extensive processes. Operating large-scale simulations by using agent-based systems has gained attention in many research areas such as biology systems  [1], [2], manufacturing systems [3] and in supply chains [4].

The application of HPC to ABM and simulation has created a rich environment for running large-scale and complex simulations. However, the complexity and heterogeneous memory of HPC systems poses challenges to the minimization of the gap between processor speed and main memory cycle time. This gap doubles every 1 to 2 years, which makes it one of the most critical challenges in the computing industry according to Machanick [5]. Managing data movement between processors and memory (known as the *memory wall* problem) is the most obvious challenge, especially in systems that deal with large amounts of heterogeneous data, for example when simulating large models. Most simulators for ABM are therefore memory bound: the agent uses memory to hold its variables (or internal state) and communication between agents. The memory wall problem has become more evident particularly in simulators using a streaming-based (data in, calculate, data out) approach to iteratively transform the memory of agents. This situation seriously impacts overall performance for large populations and scalable models since increasing amounts of data need to be moved.

To address this issue, this research proposes a method that can be applied to the simulation of complex Multi-Agent Systems (MAS) on GPUs. We demonstrate a data-aware approach to simulation experimentally using Flexible Large Scale Agent Modelling Environment for the GPU (FLAME GPU)[1](explained in section 2.3). However, the underlying model abstraction is appropriate for any streaming based MAS platform or model.

We investigate the impact of minimising data movement on performance using FLAME GPU in this paper. To minimise memory access during simulation time, this study will focus on how the required data are extracted from agent behavior during simulation and how this information can be used to reduce data movement. We extend the FLAME GPU software to demonstrate this technique using a benchmark model.

[1] http://www.flamegpu.com/

## II. BACKGROUND AND CONTEXT

### A. Improving performance in ABM

In the field of ABM, simulation by modeling individuals helps to create a natural and flexible environment for studying systems behavior, but this requires considerable computational power. Traditionally, ABM platforms use serialised algorithms in their structures to run simulations and manipulate mobile discrete agents. However, this technique limits simulation speed and model scalability [6]. This implies the need for an HPC environment or specialized workstation of parallel, or distributed, platforms [7]. Much research has focused on enhancing the performance of ABM platforms using different strategies. Distributing simulations to minimise simulation time is one such strategy, and distributed simulations of multi-agent systems can be implemented using a dedicated computing cluster [8] [9] [10] [11] or a grid [12] [13]. However, the increase in performance that is achieved by applying CPU parallelism (using distribution techniques) may be affected by a number of issues including management of communication between dynamic resource allocations and nodes, and monitoring of the state of the distributed simulation. Exploiting the shared memory parallel architecture of a GPU to run simulations has the potential to overcome many of these problems [14] [1]. To address the limitations of previous ABM platforms, FLAME GPU is designed with parallelism in mind to overcome these issues [1]. Additionally, real-time visualisation is efficient and can be directly accessed since it is also held within the GPU's memory. The result of this approach is that GPU memory bandwidth becomes the limiting factor rather than network bandwidth.

### B. Reduction of memory movement

In large parallel architecture systems, interconnections become more hierarchical; this increases the memory access gap, affecting both system latency and bandwidth [15]. Reducing data movement (data transfer between processors and system memory) in such systems, will improve overall performance. A number of techniques (and associated studies) have focused on this goal; these include: load balancing [16], [17] [18], graph partitioning [19] [20] and spatial partitioning (or spatial messaging) [1] [21]. Generally, the graph partitioning algorithm evenly divides work among computation nodes to minimize data movement. To improve performance and reduce data transfer across the system, Barrera et al. [19] used the graph partitioning technique. They automatically applied task dependency graphs during system runtime to collect information, and then used advanced graph partitioning to break the graphs into smaller parts. These partitions were used to minimize data movement across the shared memory system. To minimize data movement between processors and reduce workflow execution time, Tanaka and Tatebe [20] applied the multi-constraint graph partitioning method to the workflow directed acyclic graph (DAG) which represents task dependency. The graph partitioning method in this study helped to decrease workflow execution time by 31% and reduce the remote file access from 88% to 14% of total file access.

To reduce memory transfer in large-scale MAS, a number of data structure accelerating algorithms have been used, one of which is spatial partitioning [22]. The main aim of the spatial partitioning technique is to reduce the communication overhead in the simulation, where only subset of agents interact, reducing memory movement. This technique has notably been used in interacting systems such as swarm-based systems on GPUs [21], [23], on computing clusters [24] and on the PS3 [25]. The spatial partitioning algorithm was used to minimise the number of messages that were read by each agent based on the interaction radius of the message or particle [1].

Load balancing is another type of strategy that can be used to reduce data movement or balance compute load, especially in distributed applications. ”It minimizes the total waiting time of the resources as well as avoiding too much overload on the resources” according to Mishra [26]. There are a number of studies that focused on discussing the concept of load balancing and how to improve system performance and efficiency, such as [16], [17] and [18]. In [16], an enhanced dynamic load balancing algorithm is proposed to improve performance in grid computing, whereas [17] and [18] reviewed the implementation of a number of load balancing algorithms in cloud computing and discussed effects on cloud computing applications.

### C. An overview of FLAME GPU

Within FLAME GPU, every GPU thread represents a single agent and a (GPU) device wrapper function is used for each agent function to hide GPU memory access [27]. However, even with special techniques for hiding the cost of memory access, GPU memory bandwidth is a limited resource in large and complex models. The formal representation of an agent within the current FLAME GPU is based on the concept of a communicating X-machine (an extension to the finite state machine that includes memory). The formal definition of an X-machine describes the X-machine agents as state machines that are able to communicate with each other via messages stored in globally accessible message lists [6]. Three major components are needed to execute a model using FLAME GPU: agents, messages and layers. Agents present the agent description, messages show how the agents communicate with each other, and layers show the order of agent behaviour during the simulation. Both agents and messages have their own memory to hold agent properties and the information that needs to be passed between agents. As this paper focuses on reducing data movement, this section will show how FLAME GPU handles data movement during the simulation and how an agent's internal memory is updated.

Within FLAME GPU, each agent function (the main representation of agent behaviour) is represented by a unique GPU kernel. Using this process, global synchronization of the entire agent population is ensured after each transitional stage. Agent data in parallel threads will be stored temporarily in both the fast multiprocessor register, and shared memory. When moving data from global device memory an array of structures (AoS) is used to allow more efficient memory access for both reading and writing data. GPU memory coalescing allows for more efficient use of memory; data is consecutively accessed and fewer wide memory requests are issued [27].

An abstraction of messaging is used for internal communication between agents in FLAME GPU. Message processing within FLAME GPU supports three different techniques for reducing the transmission of data: brute force, spatially-distributed, and discrete. Each agent reads every available message in brute force messaging; to accelerate this process, shared memory is used to load messages that are accessed by agents within a group of threads. Shared memory is much faster than global memory because it is located per thread block allowing a group of threads to access the same shared memory [6]. In spatially-distributed messaging, agents can read messages within a fixed radius in a 2D or 3D continuous space. For this messaging type, FLAME GPU uses a parallel sort algorithm to reorder agents and build a matrix containing the start and end index positions of any agents within a fixed (message) radius that is sized to represent the agent's environment. After iterating through message lists within neighbouring partitions, only agents within the defined radius will be returned. Using texture memory[2]to load messages will accelerate the message reading as shared memory cannot be used in this technique because agents are in different locations and access different messages stored in different positions. This is equivalent to previous work on data structures for reducing memory transfer. Within the last type, the discrete messaging technique, shared memory or the texture cache can be used to load a 2D discrete grid of messages [1], [6].

## III. METHODOLOGY: DATA AWARE MODEL FOR AGENT REPRESENTATION

The formal representation of an agent in FLAME GPU, is based around the concept of a communicating X-Machine (a form of state machine that includes memory). The formal definition X-Machine [28] is an 8-tuple = (Σ; Γ; Q; M; Φ; F; q0; m0 )Where:

- Σ and Γ are the input and output alphabets.
- Q is the limited set of states.
- M is an infinite set called memory.
- Φ is a set of partial functions $\varphi$; each function of this type maps an input and a memory value to an output and a possibly different memory value, $\varphi$: Σ × M → Γ × M'.
- F is the next state partial function, F : Q × Φ → Q. F is often described as a state transition diagram.
- q0 is the initial state and m0 is the initial memory.

Adding the ability to X-machines to communicate with each other can be achieved by using communicating X-machines (CXM). The general definition of communicating X-machine model that is able to exchange messages is as the tuple [29]:

$$((C_i^x)i = 1..n, R)$$

Where:

- $C_i^x$ is the $i - th$ communicating X-machine in the system, and
- $R$ is a communicating relation between the $n$ X-machines.

In FLAME GPU, an agent is represented as a form of state machine that consists of: internal memory (M as in the formal definition), an agent's functions (next state partial functions, F, in the formal definition) a set of states (Q as in the formal definition) and the X-Machine agents can only communicate through messages. This can be observed in Figure 1, this state diagram represents a set of states, functions and input and out data that can be processed through these functions and the agent's memory can be updated every time step that is needed through this process. The smallest unit that can be processed by current FLAME GPU is an agent and whenever agents communicate with each other, all agent memory needs to be updated in every transition function from one state to another.

We propose an alternative representation of CXM model in which individual units of $m$ (members of the memory set M) for each agent function ($\varphi$ in Φ, in the formal definition) that can communicate using a subset of data in the messages list $r$ (members of the communicating relation set R). Focusing on a subset of this data will minimise the data movement and memory transitions. Extracting data dependencies of agents functions is the key to solve this issue. Figure 2 shows the main idea of our proposed method. Instead of reading and writing all agent memory in every state transition, the focus will be on the dependent data of each function (the subset of agent memory that has been used within each function).

## IV. AUTOMATIC DISCOVERY OF DATA DEPENDENCY FROM THE EXISTING CS MODEL

This section proposes an automatic method to extract data dependency from an existing complex-scaling (CS) model described using X-machine modelling. It translates an existing CXM model into a representation with the data-aware form described in the previous section. Customising kernel wrapper functions by reading and writing a subset of the agent will help to minimize the data movement between host and device during simulation time. An automated process has been created using flex and bison tools to parse agent function C code from FLAME GPU and produce all dependency data between

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

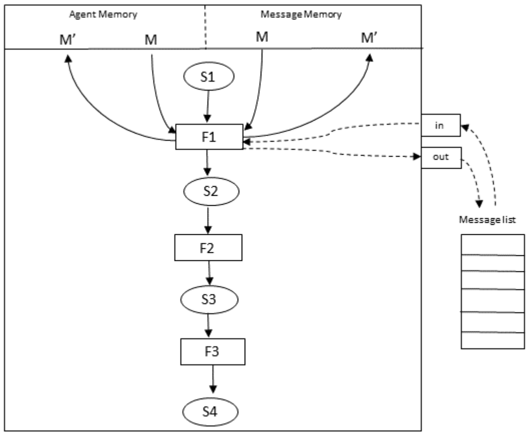


Fig. 1. Stream X-Machine Specification, M and M' represent the agent memory set before and after agent function F1 which inputs and outputs messages to the message list. [27]

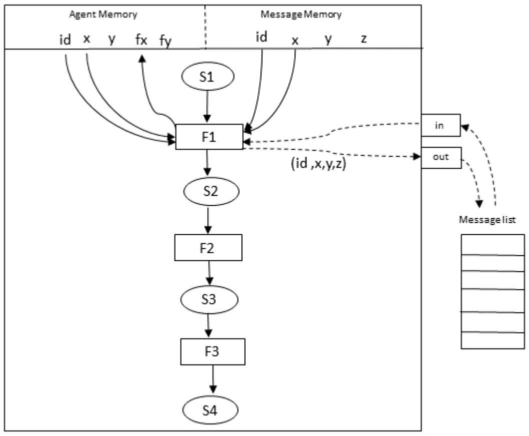


Fig. 2. The smallest unit that can be processed through transition function is individual variables of agent memory instead of an agent's full memory set to minimise data movement.

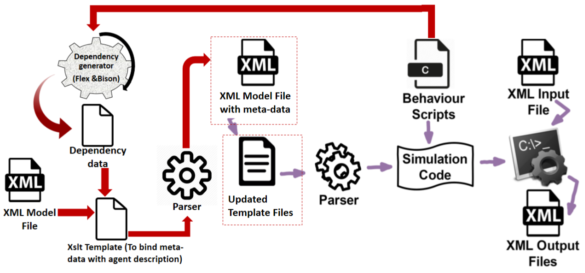


Fig. 3. Processing stages used to create the FLAME GPU runtime, showing original (purple) and additional (red) data paths.

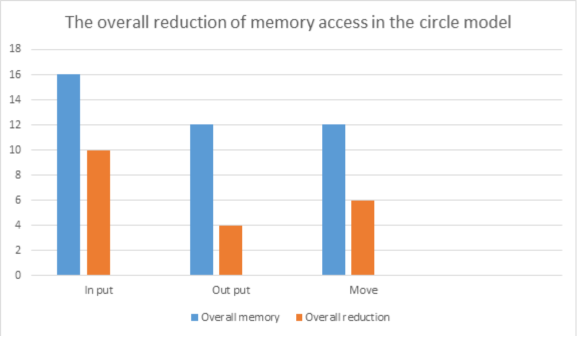


Fig. 4. The total data movement reduction of each function within circle model.

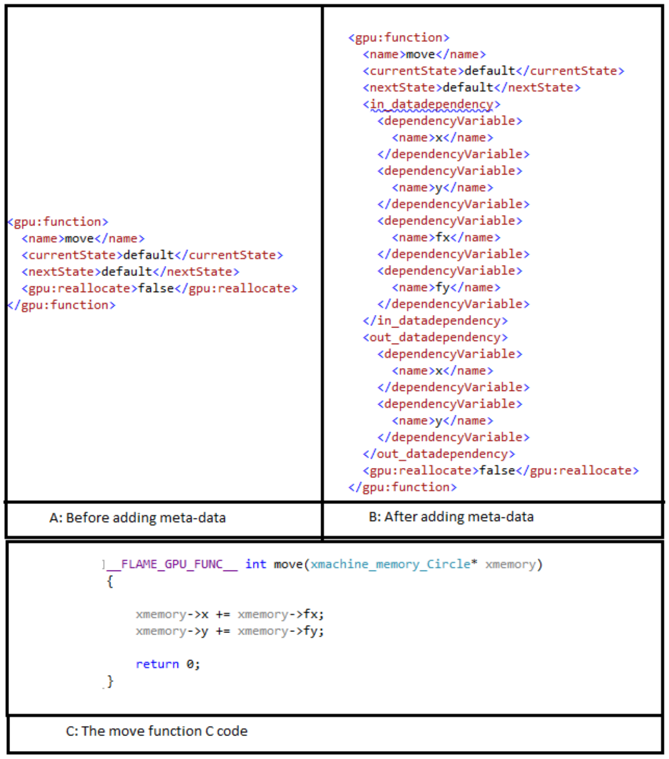
transaction functions. Extensible Stylesheet Transformations (XSLT) are then used to create the new version of the model, `XMLModelFile.xml`, which is data-aware and will reduce data movement. In the first stage, the parser will generate an XML file that consists of function names and the `In_data`, `Out_data`, and `In_messages` within each function if there are any. By combining the original `XMLModelFile.xml` with the produced file from the first stage, the second stage will generate a new `XMLModelFile.xml` using (XSLT). The output of this process is a new `XMLModelFile.xml` that contains meta information describing data dependencies. Figure 3 shows how the proposed method is linked to the current FLAME GPU. In this figure, which describes the processing stages used to create the FLAME GPU runtime, the existing data path is shown by purple arrows and the data path used in our new pre-processing stage is shown with red arrows. The dependency generator parses the behavioural function scripts and produces an XML file of data dependency. Combining this file with the XML model file using the XSLT processor will generate an XML model file with extra meta information that will help to reduce data movement. An updated XSLT template processor converts the new XML model into simulation code by customising data transactions from the discovered meta-data. This meta-data will be linked to function scripts to generate an executable simulation. There are three types of data that need to be discovered: the in-data to each function, the out-data from each function and finally the in-messages. An example of in-data and out-data is shown in figure 5: the left side (A) of the figure shows the original code of the model description while the right side (B) of the figure shows the model description after adding meta-data. This example shows the dependency data that have been extracted from the function called 'Move' within the `Circle` model [30]. Figure 4 and table I shows the total reduction of data within each function in the circle model.

## V. Apply the Changes to FLAME GPU Template to Access the Required Data

FLAME GPU generates simulations by linking XSLT templates with the function files to generate a simulation program. All agents and messages memory will be accessed during

TABLE I
THE TOTAL MEMORY ACCESS FOR EACH AGENT FUNCTION IN THE CIRCLE MODEL AND THE PERCENTAGE OF REDUCTION AFTER APPLYING OUR APPROACH

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



```
<gpu:function>
  <name>move</name>
  <currentState>default</currentState>
  <nextState>default</nextState>
  <gpu:reallocate>false</gpu:reallocate>
</gpu:function>
```

A: Before adding meta-data

```
<gpu:function>
  <name>move</name>
  <currentState>default</currentState>
  <nextState>default</nextState>
  <in_datadependency>
    <dependencyVariable>
      <name>x</name>
    </dependencyVariable>
    <dependencyVariable>
      <name>y</name>
    </dependencyVariable>
    <dependencyVariable>
      <name>fx</name>
    </dependencyVariable>
    <dependencyVariable>
      <name>fy</name>
    </dependencyVariable>
  </in_datadependency>
  <out_datadependency>
    <dependencyVariable>
      <name>x</name>
    </dependencyVariable>
    <dependencyVariable>
      <name>y</name>
    </dependencyVariable>
  </out_datadependency>
  <gpu:reallocate>false</gpu:reallocate>
</gpu:function>
```

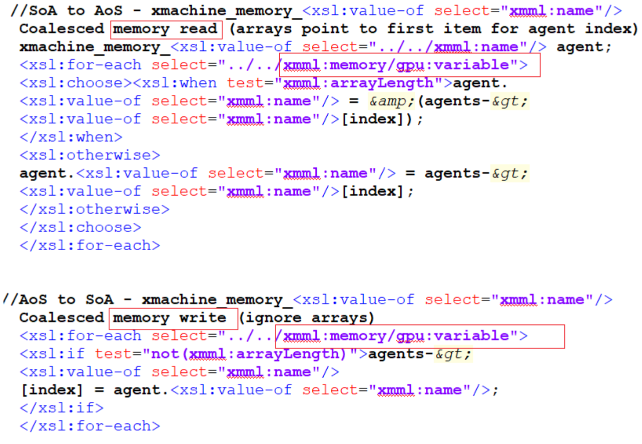
B: After adding meta-data

```
__FLAME_GPU_FUNC__ int move(xmachine_memory_Circle* xmemory)
{

    xmemory->x += xmemory->fx;
    xmemory->y += xmemory->fy;

    return 0;
}
```

C: The move function C code

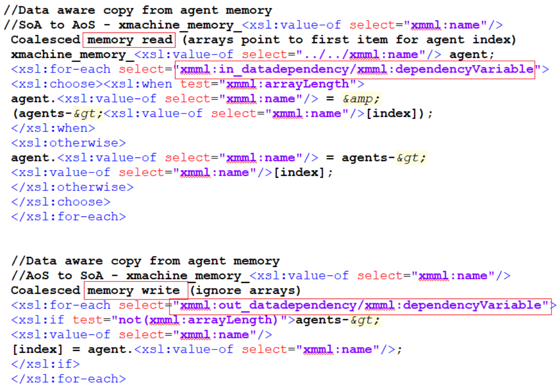
Fig. 5. A: A part of the circle model description showing function 'move'. B:The model description after adding meta-data. C: The actual body of the function 'move' from function.c file

this process using fast caches, shared memory for agent variables and texture memory for message variables. With the proposed method the templates have been modified to access only required data for both agent and messages. The original template that accesses all agent memory is shown in Figure 6 while Figure 7 shows the template code accessing a subset of this data for both reading and writing memory.



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

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

Fig. 6. The original XSLT template generating code that accessing all memory.



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

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

Fig. 7. The modified XSLT template that generates code accessing required data only.

## VI. RESULTS

To evaluate the benefits of using the proposed system, this section will show the comparison of results between the current FLAME GPU and the modified version using the benchmark model that was proposed by [31]. The benchmark model is based on the concept of particle-based simulation and accepts input parameters that control both system scalability and agent homogeneity. For system scalability, the population size for each agent type will be increased. In agent homogeneity, the focus will be on increasing the complexity for both individuals (by increasing communication) and the overall population (by increasing diversity of agent type). Three different benchmarks were used to examine the performance efficiency for both systems: scalability, divergence within the population and divergence within an agent. The machine used for benchmarking both versions uses NVIDIA TITAN Xp graphics card with 3840 CUDA cores and 12 GB of memory. An average for running each experiment was 10 times for each sample.

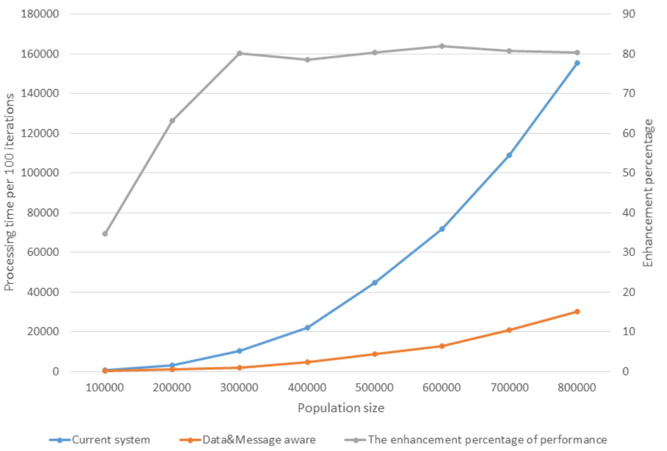


Fig. 8. Comparison of average execution time against population size, showing unmodified (blue) and modified (orange) FLAME GPU; and rate of improvement (grey)

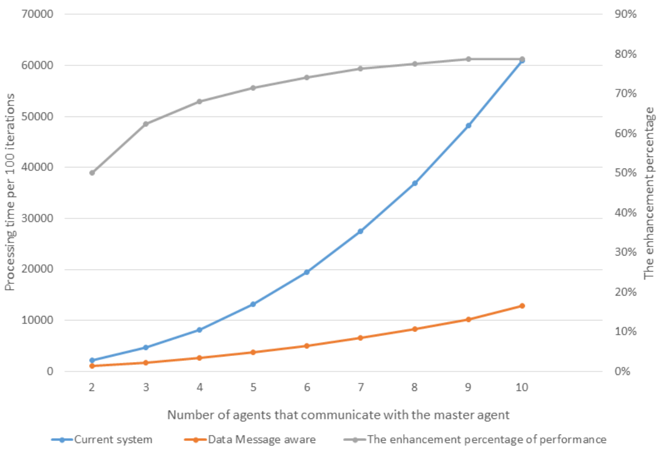


Fig. 9. Comparison of processing time against number of communicating agents (slave-to-master), showing unmodified agents (blue) and modified (orange) FLAME GPU; and rate-of-improvement (grey).

### A. Scalability

This benchmark measures the scalability of the performance of both systems. The population size of each agent type starts with 100,000 agents and ends with 800,000 agents. This benchmark is based on the same example that was used by Alzahrani et al. [31]. It is representative of scaling the population size of the model and the simulation was performed for 100 iterations. In Figure 8 the proposed method (orange line) shows significant speed improvements when compared to current FLAME GPU ( blue line). The grey line within the same figure shows the percentage of performance enhancement, with population size equal to 300,000 and above the average of improvement reaches 80%. This shows the optimal utilization of GPU memory by FLAME GPU ”which provides a good balance between a large number of threads required to hide memory access latencies with the limited register availability of the underlying architecture.” [32]

### B. Performance Impact on Agent Complexity

The main concept of this experiment is to observe the effects of divergent behaviour (within an agent) on the execution time for both systems. Based on the same example that is used by the benchmark model within [31], this benchmark is representative of increasing the individual complexity of an agent, and that means more functions in each layer every cycle. As the function layers represent the control flow of simulation processes in FLAME GPU, adding more agent functions every time will increase the number of layers in each cycle (as functions of the same agent need to be processed in sequential order) and that will lead to increasing the execution time of each iteration. The increase in execution time can be observed in Figure 9 for both versions (current version with the blue line and modified one with orange line) with significant improvements of execution time when using the proposed system. The improvement rate can be observed in the grey line in the same figure, and with more divergence within an agent, the modified system showed more time reduction in simulation execution time compared with the current system. The population size that has been used in this benchmark is 100,000 for each type of agent, and each simulation was run for 100 iterations using the same environment size.

### C. Performance Impact on Population Complexity

Observing the system performance while increasing population complexity will be the focus of this benchmark. This experiment starts with a simple model containing three types of agent, ten agent functions and three kind of message and ends with 30 agent types, 100 agent functions, and 30 message types. The grey line in Figure 10 shows the amount of improvement in the execution time that the proposed system achieved over the current system. The execution time that has been performed in both systems can be observed in the same figure, the blue line represents the current FLAME GPU, and the orange line shows the system using the proposed method. Each simulation in this benchmark was run for 100 iterations using the same environment size and the population size of each agent type was 100,000 agents.

## VII. Conclusions

In this paper, we have shown the impact of minimising data movement on performance for complex simulation models using FLAME GPU. Three stages have been carried out within this work, to reduce data movement and examine the effects on the system. The first stage focused on designing and implementing a dependency parser to analyse and integrate data dependency within Complex System (CS) models. The second step showed how the FLAME GPU framework was modified to minimise data movement based on discovered data dependencies. In the third stage, a benchmark model has been used to experimentally evaluate the performance of decreasing the data movement through a benchmark model.

Three benchmark experiments have been used to evaluate the overall performance of the new system. These experiments

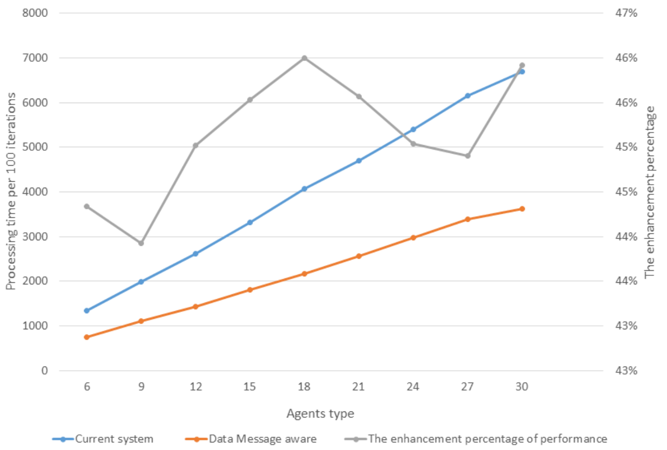


Fig. 10. Comparison of average execution time against population divergence, showing unmodified of using current (blue) and modified (orange) FLAME GPU; and the rate of improvement (grey).

focused on measuring the ability for the new system to reduce simulation execution time under specific criteria (scalability and system homogeneity). Comparing the benchmark results of the current and new system show that reducing data movement within CS simulation improves overall performance. The scaling population size experiment for both systems showed that the new method helped to reduce execution time by approximately 80% and tends to stabilise around this percentage as population size increases. A significant improvement has resulted from using the proposed method within the divergence benchmark. Execution time was reduced by 70% when running the agent divergence benchmark and by around 45% while examining the population divergence experiment.

Work is ongoing to evaluate the proposed method using different models such as the circle model [30] and the Keratinocyte (cell) model [33]. For the next step, comparing FLAME GPU with other ABM platforms using this approach is needed. Our plan is to apply the data-aware approach on other platforms (OpenMP applications) and compare the results with FLAME GPU using the same models.

## References

[1] P. Richmond, D. Walker, S. Coakley, and D. Romano, “High performance cellular level agent-based simulation with flame for the gpu,” *Briefings in bioinformatics*, vol. 11, no. 3, pp. 334–347, 2010.

[2] A. de Paiva Oliveira and P. Richmond, “Feasibility study of multi-agent simulation at the cellular level with flame gpu.” in *FLAIRS Conference*, 2016, pp. 398–403.

[3] J. Barbosa and P. Leitão, “Simulation of multi-agent manufacturing systems using agent-based modelling platforms,” in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. IEEE, 2011, pp. 477–482.

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

[5] P. Machanick, “Approaches to addressing the memory wall,” Tech. rep., School of IT and Electrical Engineering, University of Queensland, Tech. Rep., 2002.

[6] P. Richmond and M. K. Chimeh, “Flame gpu: Complex system simulation framework,” in *High Performance Computing & Simulation (HPCS), 2017 International Conference on*. IEEE, 2017, pp. 11–17.

[7] S. Abar, G. K. Theodoropoulos, P. Lemarinier, and G. M. O’Hare, “Agent based modelling and simulation tools: a review of the state-of-art software,” *Computer Science Review*, vol. 24, pp. 13–33, 2017.

[8] V. Suryanarayanan, G. Theodoropoulos, and M. Lees, “Pdes-mas: Distributed simulation of multi-agent systems,” *Procedia Computer Science*, vol. 18, pp. 671–681, 2013.

[9] V. Suryanarayanan and G. Theodoropoulos, “Synchronised range queries in distributed simulations of multiagent systems,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 23, no. 4, p. 25, 2013.

[10] R. Minson and G. K. Theodoropoulos, “Distributing repast agent-based simulations with hla,” *Concurrency and Computation: Practice and Experience*, vol. 20, no. 10, pp. 1225–1256, 2008.

[11] G. Theodoropoulos and B. Logan, “The distributed simulation of agent-based systems,” in *IEEE Proceedings Journal, Special Issue on Agent-Oriented Software Approaches in Distributed Modeling and Simulation*. Citeseer, 2001.

[12] G. Theodoropoulos, Y. Zhang, D. Chen, R. Minson, S. J. Turner, W. Cai, Y. Xie, and B. Logan, “Large scale distributed simulation on the grid,” in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 2. IEEE, 2006, pp. 63–63.

[13] M. Pipattanasomporn, H. Feroze, and S. Rahman, “Multi-agent systems in a distributed smart grid: Design and implementation,” in *Power Systems Conference and Exposition, 2009. PSCE’09. IEEE/PES*. IEEE, 2009, pp. 1–8.

[14] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe, “Using gpu for multi-agent multi-scale simulations,” in *Distributed computing and artificial intelligence*. Springer, 2012, pp. 197–204.

[15] A. Plaat, H. Bal, and R. Hofman, “Bandwidth and latency sensitivity of parallel applications in a wide-area system,” 1998.

[16] P. K. Srivastava, S. Gupta, and D. S. Yadav, “Improving performance in load balancing problem on the grid computing system,” *International Journal of Computer Applications (0975–8887) Volume*, 2011.

[17] R. Kaur and P. Luthra, “Load balancing in cloud computing,” in *Proceedings of International Conference on Recent Trends in Information, Telecommunication and Computing, ITC*. Citeseer, 2012.

[18] F. F. Kherani and J. Vania, “Load balancing in cloud computing,” 2014.

[19] I. S. Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas, “Reducing data movement on large shared memory systems by exploiting computation dependencies,” 2018.

[20] M. Tanaka and O. Tatebe, “Workflow scheduling to minimize data movement using multi-constraint graph partitioning,” in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 65–72.

[21] P. Richmond and D. Romano, “Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu,” in *Proceedings International Workshop on Supervisualisation*, 2008.

[22] B. Li and R. Mukundan, “A comparative analysis of spatial partitioning methods for large-scale, real-time crowd simulation,” 2013.

[23] U. Erra, R. De Chiara, V. Scarano, and M. Tatafiore, “Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance,” *Proceedings of Vision, Modeling and Visualization 2004 (VMV)*, 2004.

[24] P. Goyal, S. Kumari, S. Sharma, D. Kumar, V. Kishore, S. Balasubramaniam, and N. Goyal, “A fast, scalable slink algorithm for commodity cluster computing exploiting spatial locality,” in *2016 IEEE 18th International Conference on High-Performance Computing and Communications, IEEE 14th International Conference on Smart City, and IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2016, pp. 268–275.

[25] C. Reynolds, “Big fast crowds on ps3,” in *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. ACM, 2006, pp. 113–121.

[26] N. K. Mishra and N. Mishra, “Load balancing techniques: Need, objectives and major challenges in cloud computing-a systematic review,” *International Journal of Computer Applications*, vol. 131, no. 18, 2015.

[27] P. Richmond, S. Coakley, and 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*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1125–1126.

[28] A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan, “Communicating stream x-machines systems are no more than x-

machines," *Journal of Universal Computer Science*, vol. 5, no. 9, pp. 494–507, 1999.

[29] S. Coakley, R. Smallwood, and M. Holcombe, "Using x-machines as a formal basis for describing agents in agent-based modelling," *Simulation Series*, vol. 38, no. 2, p. 33, 2006.

[30] P. Richmond and D. Romano, "Template-driven agent-based modeling and simulation with cuda," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 313–324.

[31] E. Alzahrani, P. Richmond, and A. J. Simons, "A formula-driven scalable benchmark model for abm, applied to flame gpu," in *European Conference on Parallel Processing*. Springer, 2017, pp. 703–714.

[32] S. Coakley, P. Richmond, M. Gheorghe, S. Chin, D. Worth, M. Holcombe, and C. Greenough, "Large-scale simulations with flame," in *Intelligent Agents in Data-intensive Computing*. Springer, 2016, pp. 123–142.

[33] T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood, and S. MacNeil, "An integrated systems biology approach to understanding the rules of keratinocyte colony formation," *Journal of the Royal Society Interface*, vol. 4, no. 17, pp. 1077–1092, 2007.