DE GRUYTER OPEN
1862
RIGA TECHNICAL UNIVERSITY
Information Technology and Management Science
ISSN 2255-9094 (online)
ISSN 2255-9086 (print)
December 2017, vol. 20, pp. 97–102
doi: 10.1515/itms-2017-0017
https://www.degruyter.com/view/j/itms

An Application of Graphics Processing Units to Geosimulation of Collective Crowd Behaviour

Jānis Cjoskāns[1], Arnis Lektauers[2]
[1] *Rural Support Service of the Republic of Latvia, Latvia*, [2] *Riga Technical University, Latvia*

***Abstract* – The goal of the paper is to assess the ways for computational performance and efficiency improvement of collective crowd behaviour simulation by using parallel computing methods implemented on graphics processing unit (GPU). To perform an experimental evaluation of benefits of parallel computing, a new GPU-based simulator prototype is proposed and the runtime performance is analysed. Based on practical examples of pedestrian dynamics geosimulation, the obtained performance measurements are compared to several other available multi-agent simulation tools to determine the efficiency of the proposed simulator, as well as to provide generic guidelines for the efficiency improvements of the parallel simulation of collective crowd behaviour.**

***Keywords* – Crowd behaviour, geosimulation, GPU computing.**

I. Introduction

Modelling and simulation of collective behaviour today is important in many areas – including ecology, biology, sociology, as well as urban planning. A valuable approach to simulation of collective behaviour is the use of geosimulation. However, practical implementation of such models is often difficult because there is a need to define each model element as a separate entity, as well as to describe complex spatial relations between the elements. This leads to complex models containing a large number of model elements, therefore requiring significant computational resources during simulation execution.

The goal of the paper is to assess the ways for computational performance and efficiency improvement of collective crowd behaviour simulation by using parallel computing methods implemented on graphics processing unit (GPU).

To perform an experimental evaluation of benefits of parallel computing, a new GPU-based simulator prototype is proposed and the runtime performance is analysed. Based on practical examples of pedestrian dynamics geosimulation, the obtained performance measurements are compared to several other available multi-agent simulation tools to determine the efficiency of the proposed simulator, as well as to provide generic guidelines for the efficiency improvements of the parallel simulation of collective crowd behaviour.

A. GPU-based Parallel Computing

Modern graphics processing units (GPUs) can be programmed using a wide variety of toolchains and frameworks with a varying level of abstraction. One of the dominating toolchains for general purpose GPU (GPGPU) computing is NVIDIA CUDA [1] – C language based toolchain and development framework for GPUs manufactured by NVIDIA company. CUDA is intended for development of parallel computing solutions, using the so-called compute kernels, intended to be run on GPU hardware. Compared to more cross-hardware toolchains, such as OpenCL [2], which provide capability to develop programs that can be executed on a wide variety of compatible computational devices (CPUs, GPUs, FPGAs etc.) without significant changes in code, CUDA provides lower level access to underlying hardware architecture, giving more control over device-specific functions and, as a result, higher theoretical performance [3], [4]. However, code written in CUDA is executable only on NVIDIA GPUs. CUDA toolchain also provides mature integrated debugging and profiling toolset, allowing for efficient code debugging and optimisation. CUDA also provides libraries for solving typical problems (linear algebra, Fourier transformation, graph analytics etc.) as a drop-in acceleration for the existing code with minimal code changes.

II. Geosimulation in the Context of Collective Crowd Behaviour

Geosimulation is a general concept that specialises on solving of complex geospatial problems using micro-simulation of spatially related automata [5] mainly focusing on an integrated use of cellular automata and agent-based models.

A. Geographic Automata

Geographic automata (GA) define certain behaviour that can be described by their internal state and spatial relation to other automata and their surrounding environment. Geographic automata, by their formal definition presented in [5], are based on and extend the definition of finite automata by introducing functionality to enable explicit spatial behaviour and spatial awareness. Formally, finite automaton can be presented as a set of finite internal states $S = \{S_1, S_2, \dots, S_N\}$, external information I and a set of transition rules $T\colon (S_t, I_t) \rightarrow S_{t+1}$ at any time moment t [5]:

$$A \sim (S, T) \quad (1)$$

By this definition, geoautomaton can change its internal state as well as spatial position according to defined state transition rules, both non-spatial and spatial. Geographic automata also provide functionality to define a set of neighbours and neighbourhood rules between automata, enabling application of state transition rules according to inter-automata dependencies (neighbourhoods) that are dynamically changing in space and time.

B. Geographic Automata Systems

Geographic Automata Systems (GASs) can be considered as a combination of multi-agent systems (MASs) and cellular automata (CA) where each geoautomaton can be simulated as a

©2017 Jānis Cjoskāns, Arnis Lektauers.
This is an open access article licensed under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), in the manner agreed with De Gruyter Open.

separate agent [6]. This approach allows for micro-level simulation and bottom-up modelling of complex phenomena, enabling observation of emergent properties within a system at a higher level of scale, which are not directly predictable from constituent parts [7], [8]. The emergence of new properties, thus, is related to the observation scale and context, and often can be observed and identified only within large groups of interacting agents. This requirement for a large number of agents, together with agent complexity, is often the limiting factor for simulating such models. For GA models containing a large number of interdependent agents, with their explicit requirement to process a large number of interdependent spatial relations, achieving adequate simulation performance can become a serious implementation challenge, as spatial relation operations are often computationally intensive. For some models, this can be at least partially mitigated by reduction of model detail, at a risk of oversimplification that can lead to unexpected and invalid results.

One of the technical solutions to enable larger models and thus ability to observe larger-scale emergent properties without sacrificing model detail is to use high-performance computing (HPC) for parallel simulation. Since agents in MASs can be considered autonomous and capable of independent decision making [6], in many models agent states can be calculated in parallel and results integrated in the global model state afterwards. It should be noted though that in some models parallel processing is not possible or is severely limited in application, for example, due to requirements for specific agent processing order [9], so it is critical to determine these factors during development of model and simulation requirement specification.

III. Parallelization of Collective Behaviour Simulation

In order to assess the potential benefits of parallel crowd behaviour simulation, the CPU-based open source microscopic crowd simulation library PEDSIM [10] was adapted for parallel execution by using the CUDA 8.0 toolkit. The performance for different implementations was analysed and compared between implementations. All the development was done using Visual Studio 2015 and CUDA 8.0 for the 64-bit Windows platform.

Since CUDA as of version 8.0 is mostly tended for the C language and support for C++ constructs is limited, as the first step of adaption, the initial PEDSIM implementation was rewritten using the C language constructs (structures and functions instead of classes and methods) to derive a C library implementing the necessary PEDSIM functionality. To facilitate the development, all the functions and data structures that did not directly require different implementation for parallel simulation, were left unchanged between implementations, providing for easier code maintenance. As a result, the code was not fully optimised for GPU implementation and performance obtained was lower than theoretically achievable – this was considered an acceptable trade-off for significantly easier and faster development process.

Initial version of the library uses a single CPU thread to perform simulation (CPU method), providing strictly sequential simulation processing. For purposes of the parallel simulation, two easily available parallel computing approaches were implemented – using GPU cores (further referenced as GPU method) and using all available CPU cores (further referenced as MCPU method). The most notable difference between both implementations is the agent processing sequence – in the GPU method, a single core processes a single agent, and all cores process their own agents in parallel, as limited by a number of available cores in a device. If the number of available cores is lower than the agent count, the rest of agents are put in a "waiting" state, to be processed as cores become available, but a single core still processes a single agent. In the MCPU implementation, the agent list is divided equally between CPU threads (created for each of available physical CPU cores) in blocks and each CPU thread processes its agent block sequentially, but multiple threads process their blocks in parallel. It should be noted that if the agent count is lower than the available core (CPU or GPU) count, both methods perform identically, with each core processing a single agent, all cores processing fully in parallel. Implementations differ mostly to avoid sequential processing in the GPU cores (GPU execution thread management is done automatically with minimal overhead) and lower the overhead associated with launching and managing execution of the threads on CPU.

A. Crowd Agent Model

PEDSIM library implements the concept of a social force model as described in [11] and [12]. The behaviour of each agent $j \in J$ is described by three vectorial force functions – a motivation function to move towards the destination f_0, the force function f_j of an interaction with other agents and the force function f_s of obstacles S:

$$\frac{\mathrm{d}\vec{v}}{\mathrm{d}t} = A_m f_0 + A_i \sum_{i \neq j} f_j + A_s \sum_S f_s, \quad (2)$$

where $\vec{v}$ – the agent speed at time moment t;

A_m – the weight of motivation force;

A_i – the weight of interaction force;

A_s – the weight of interaction with obstacles.

The social force model requires calculation of the psychological forces exerted on each agent by other agents. The simplest way to calculate these forces is to calculate forces between agent and every other agent in the model; however, for large models this is impractical and computationally expensive. Considering that the forces from the agents that are far away are very low and their impact is negligible, the library includes only the agents that are nearby (located not further than eight meters away) and calculates the forces only for those agents.

B. Spatial Index for Crowd Simulation

To facilitate faster neighbourhood search and avoid unnecessary distance calculations, a spatial index is used. Originally library implements the Quad-Tree spatial indexing approach [13] with dynamically allocated and removed elements. This structure and its existing implementation were considered as not well suited for parallel processing as it uses intensive global memory allocation (which would require excessive thread synchronisation, reducing gains from parallel execution) and for large models would tend to exceed

maximum technically possible recursion depth [14]; spatial index was replaced with a simple static single level grid index.

The spatial index divides model space into rectangular tiles and associates each tile with agents spatially within it as shown in Fig. 1, allowing fast search of agents within a specific spatial region. Since tile spatial positions are static and known, with simple arithmetic operations it is possible to easily determine neighbouring tiles and from their associated agent lists quickly perform rough neighbourhood search for any agent within a model. Combined with secondary filtering by 8 meter distance from an agent, this theoretically allows for considerably faster force calculation. For code simplicity and performance reasons it was assumed that each agent is described only as a simple point and as such could only belong to a single tile. This simplification, however, has no impact on precision in this case, since the spatial index searches are performed by a larger distance (10 meters) around an agent.

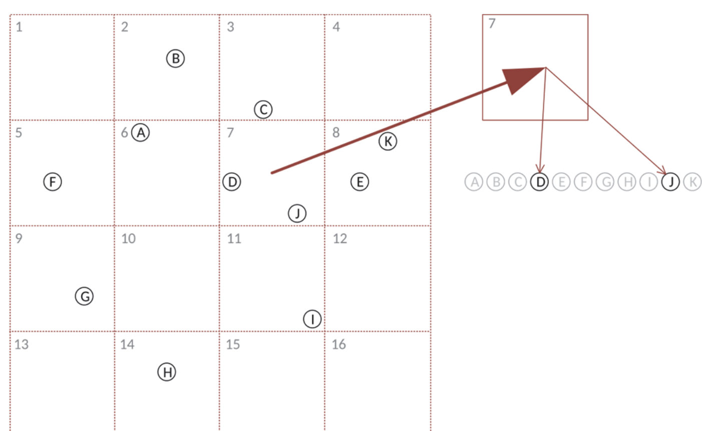

Fig. 1. An example of static 4 × 4 single-level spatial index.

Spatial index implementation for CPU and MCPU environments is rather simple – after each simulation step, for each agent its spatial position is compared with previously known tile and if an agent has left tile boundary, an agent is removed from previous tile and added to tile corresponding to its current spatial position. This operation, however fast it is on CPU, requires sequential agent processing and as such is not suitable for GPU implementation. For this reason, for GPU spatial index maintenance is implemented as a parallel two-step process, with the first step evaluating all spatial tiles in parallel, removing agents that are no longer within their previous tile. This process requires parallel thread launch for each spatial tile and evaluation of each agent within a tile agent list. When all tiles have removed those agents from their agent list, the second step is performed – also in parallel for each tile – which iterates through all agents within a model and if an agent was removed in the first step and is within a current tile, it is added to the agent list for this tile. This process requires evaluation of each agent within a model for each tile and even if tiles are processed in parallel is somewhat inefficient, albeit still much faster than simple sequential processing on a single GPU core.

IV. Simulation Performance Analysis

The simulation performance comparison was performed using a model of a 600-meter long, 100-meter wide tunnel, narrowing to 90 meters in the middle. On both sides of the tunnel, two groups of agents were placed, starting at the 160-meter boundary between the two groups, on each side half of the total number of agents, in rows of 50 agents per row. Each individual agent's goal is to reach the destination located on the opposite side of the tunnel; simulation was finished as soon as all agents reached their destinations. Three models with the only difference being the number of agents – 1000, 5000 and 10000 – were used to evaluate implementation scalability (Fig. 2). For each model, performance measurements were obtained with all three of the implemented simulation methods – CPU, GPU and MCPU. For each method, simulations were performed using two different precision floating point data types (single precision 32-bit FLOAT and double precision 64-bit DOUBLE) to determine the performance overhead of using floating point data types.

All the performance measurements were performed on two separate desktop computers with 64-bit Windows 10 operating system with configurations outlined in Table I. To reduce the impact of the operating system background processes, for each model and configuration three separate simulation runs were performed and the results between the runs were averaged to produce the final value.

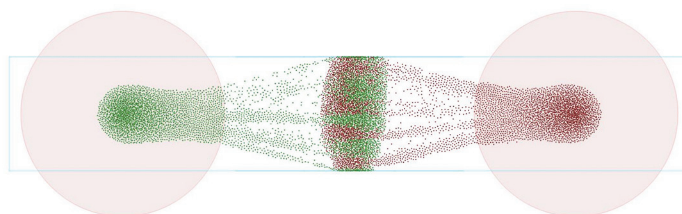

Fig. 2. The positions of crowd agents after 1020 simulation steps.

TABLE I
Computer Configurations Used for Analysis of Simulation Performance

No.	CPU	CPU cores	Graphics adapter	GPU cores
1	Intel Core i3-2120	2	GeForce GTX 950	768
2	Intel Core i7-3770	4	GeForce GT 710	192

Since a simulator was developed for performance measurement, it was explicitly adapted for execution time measurement and logging. All time periods were measured using Windows API function `QueryPerformanceCounter` with sub-microsecond resolution and logged to a separate file for further processing. Time periods were later processed to microsecond resolution, to give comparable precision on both tested configurations. For method comparison, performance score was calculated as acceleration – a ratio between simulated model time and real time used to simulate the corresponding model.

As can be seen from Table II, for 1000 agent models, single-precision MCPU method of configuration 2 scores highest in overall performance for this model size; however, for configuration 1, which has weakest CPU but more powerful GPU, single-precision GPU method scores higher than any of CPU-based methods.

TABLE II
CROWD SIMULATION PERFORMANCE MEASUREMENT RESULTS

Method	Agent count	Data type	Simulated time, seconds	Execution time, seconds		Acceleration, times	
				Config 1	Config 2	Config 1	Config 2
CPU	1000	Double	288.8	48.4	37.2	6.0	7.8
CPU	1000	Float	309.6	45.8	33.8	6.8	9.2
GPU	1000	Double	300.4	26.4	54.6	11.4	5.5
GPU	1000	Float	294.4	15.9	28.5	18.5	10.3
MCPU	1000	Double	288.8	25.9	13.3	11.2	21.8
MCPU	1000	Float	309.6	25.2	12.8	12.3	24.2
CPU	5000	Double	584.4	796.1	656.8	0.7	0.9
CPU	5000	Float	575.2	688.4	542.3	0.8	1.1
GPU	5000	Double	616.4	294.9	1154.3	2.1	0.5
GPU	5000	Float	591.6	133.5	441.9	4.4	1.3
MCPU	5000	Double	584.4	384.5	203.8	1.5	2.9
MCPU	5000	Float	575.2	344.6	168.0	1.7	3.4
CPU	10000	Double	1098.4	4495.5	3672.0	0.2	0.3
CPU	10000	Float	1162.4	4384.0	3475.1	0.3	0.3
GPU	10000	Double	1216.0	1591.7	7490.5	0.8	0.2
GPU	10000	Float	1058.8	526.5	2333.9	2.0	0.5
MCPU	10000	Double	1098.4	2072.1	982.8	0.5	1.1
MCPU	10000	Float	1162.4	2121.8	970.2	0.5	1.2

For 5000 agent models, single precision GPU score for configuration 1 takes leadership, with single-precision MCPU score for configuration 2 taking second place, the same being true for 10000 agent models, with a score gap between the first and the second positions becoming more pronounced showing that with an increase in model size, overheads associated with GPU execution have less impact on overall performance, allowing GPU method to scale better with increasing model size. This should be taken into account when deciding on the actual simulation method for a real model – impact from parallel simulation overhead for small models might negate any possible performance gains and so the model size must be taken into account during this choice.

All methods show visible performance gains from using single-precision data types – which is to be expected – but it should be noted that actual performance drop for GPU based models is much more pronounced, when compared to both CPU-based methods, and becoming more with an increase in the model size. This fact shows impact of differences on GPU and CPU core architecture – if CPU cores usually (with very few exceptions) contain a double precision arithmetic unit per core, GPUs contain a single double precision unit per multiple cores – in this case, according to the corresponding GPU documentation, a single unit per 32 cores for configuration 1, and a single unit per 24 cores for configuration 2, so theoretical throughput for double precision arithmetic is up to 24–32 times lower than single precision. In practice, however, in this case performance drop is lower due to other non-arithmetic operations and other overheads offsetting this. This aspect must also be considered during model development – a decision to use a double-precision data type must be justified by practical necessity. In this case, since a distance unit is meter, a single-precision data type corresponding to the IEEE 754 standard [15] allows for figures up to $\pm 10^4$ with precision at least 10^{-2} – it is clear that there is no need in this model for higher precision or larger numbers, so a single-precision type can be used without significantly affecting the results. It should be noted, however, that, as shown by results in Table II, simulated time to achieve a model goal is dependent on a data type, so there is visible, if small, difference in results.

The results also show that the difference in simulated time is also dependent on the method used – CPU and MCPU methods both show equal time for each model size, but GPU method differs visibly even when the same data types are used. This is due to the fact that although in theory both CPU and GPU implement arithmetic according to IEEE 754 standard, the standard is loose on technical implementation details, so each manufacturer can implement arithmetic in a slightly different way, leading to small differences in results, which are becoming more visible in iterative calculations, as in this case. This must mostly be considered when comparing simulation output data – although for all methods the results are likely to be principally correct, it can turn out to be difficult to compare results between them, especially for mathematically complex or long-running models where these small differences accumulate. In this case, this shows another point – since for CPU and MCPU simulated times are equal, it can be assumed that sequential and parallel implementations perform exactly alike on the same hardware and so the order of agent processing sequence is not essential – which is also observed by comparing actual simulation output data.

Simulator implementation can also be used to collect more detailed performance data, splitting each step time into times for each sub-step used to calculate a single simulation step.

Figure 3 shows time in microseconds for most computationally complex steps – force calculation – for GPU method. For clarity, results for other methods are not shown, but they follow the same pattern with most differences being at the vertical scale.

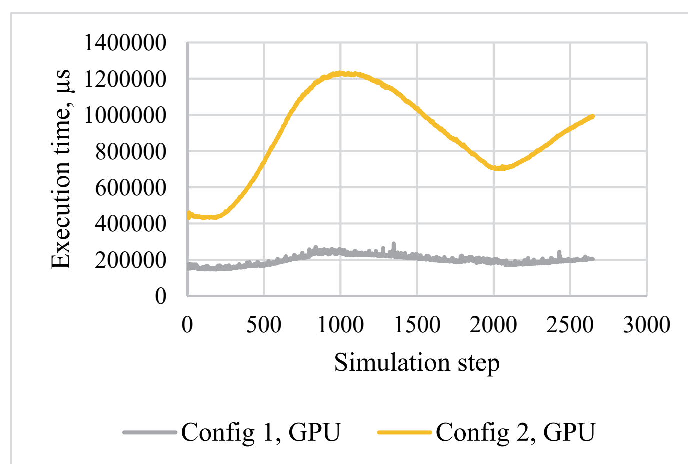

Fig. 3. Execution time of static 4 × 4 single-level spatial index.

It can be observed that force calculation time is not constant but varies during simulation. After correlating calculation time with simulation results, it is clear that force calculation time is dependent on agent density – after initial placement in rows next to each other, pedestrians initially scatter to their “comfortable” distance from each other while also moving forward, gradually reducing density up to simulation step 165 (Fig. 4). After this step agent density is gradually increasing again because both opposing groups are closing in to each other, forcing agents to “work” more in search for ways around other agents, forming trapezoidal lanes through crowd. This increase in density – and thus calculation time – continues approximately to simulation step 1020, when lanes have stabilised and both groups are moving past each other. Density starts to gradually decrease with more agents passing through crowd and approaching destination, up to step 2040, when both groups mostly have passed each other and agents start to concentrate around destination, forming two smaller crowds near each destination and forcing density and calculation time to increase.

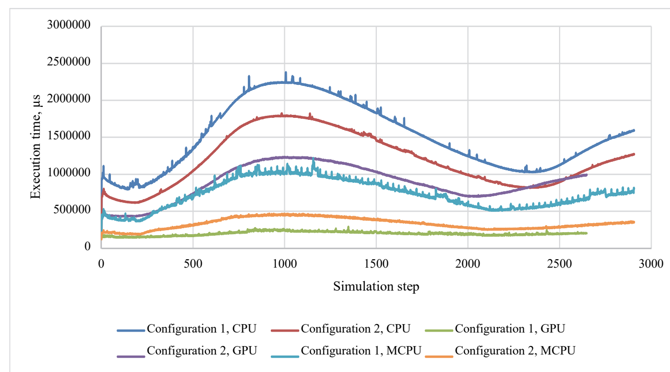

Fig. 4. Time dynamics of a single simulation step for a model containing 10000 agents.

This variation in calculation time according to density is mostly the result of the need to calculate forces exerted on each agent by other agents in its vicinity, so the higher the density of the agents, the more agents are nearby, the more forces must be calculated. As mentioned before, PEDSIM limits forces calculated to agents within distance limit of eight meters and can use a spatial index to reduce a list of agents for which distance is calculated. Since it is clear that the spatial index itself requires maintenance (constantly changing agent positions require updates to the spatial index after every simulation step) and is associated with overheads, it is advisable to determine efficiency of current spatial index implementation.

To evaluate spatial index performance, two metrics were used – spatial index maintenance time and performance gained by using the spatial index. To measure an increase in force calculation times, initial 32 × 32 spatial index was substituted by 1 × 1, i.e., a single tile, essentially disabling spatial index assistance in the force calculation, but leaving rest of the code for spatial index search and maintenance in use. It was observed that for 1000-agent models, gains from using the spatial index were negligible, if any. With an increase in model size, gains from the spatial index were becoming more pronounced, resulting in up to two times lower calculation times for 10000 agent models. It can also be noted that for more powerful CPU and GPU configurations, a spatial index gain is lower, suggesting that with enough computing power distance calculation for every agent is fast enough to offset overheads from spatial index maintenance and use. This means that benefit from the spatial index will still be available, albeit will become more pronounced with even a larger number of agents.

Spatial index maintenance overhead for both index sizes was also measured and compared. For CPU and MCPU methods, use of 1 × 1 index resulted in a drop in maintenance times to half of that observed for 32 × 32 index and became constant, which was expected. For GPU method situation was reversed – maintenance times increased more than twofold. This increase is due to fact that, even if no agent ever leaves or enters another tile, maintenance code still has to go through each tile’s agent list twice to verify that. Since there is only one tile, with a large agent list (containing all the agents in the model), this forces single thread on GPU to scan a full agent list twice, which leads to a large waste of computational resources on a weaker GPU core. This is one of the significant mistakes that can be made while developing a code meant for both sequential and parallel execution – assuming that parallel implementation will behave like sequential implementation in every situation, without taking into account specifics of parallel implementation and leading to unexpected sequential execution on GPU.

The final step – output of simulator results – was, as expected, linearly dependent on agent count for all methods. However, since simulation results were written to file and GPU does not have access to the file system, for the GPU method an additional step was required – copying agent data from the GPU-accessible memory to the CPU-accessible memory. This operation is done through the peripheral bus and requires additional time that also grows with an increase in the amount of the transferrable data. For the largest model with the slowest GPU, this time approached nine milliseconds for each step. Since in this case this transfer was unavoidable, its impact could be considered acceptable, but overheads were noticeable enough as to suggest that they should be avoided if possible. For instance, copying agent data from GPU to CPU for purposes of simply displaying them on a screen might be counterproductive since it might be possible to display them directly from GPU memory; however, this also might lead to unnecessary complexity if copying overhead could be tolerated. By copying agent data from CPU to GPU and back, it is also possible to perform simulation for models larger than GPU memory, with acceptable performance penalty, so careful evaluation on a case-by-case basis is needed.

V. Conclusion

In general, the results indicate that parallel computing solutions are very valuable tools for increasing the performance of simulation. Results also indicate that GPUs can be considered a significant alternative to expensive high-performance CPUs. GPU parallel computing solutions also have potential to scale better for larger model sizes even considering their usually limited available memory. However, when flexibility in an execution platform is needed, parallel computing using all available CPU cores is relatively easy to implement and readily available, since modern CPUs, even entry-level, usually contain at least 2–4 powerful cores. It

should be noted, as shown in case of spatial index performance, that implementing and optimising multiple simulation methods might lead to code base divergence and need to support multiple versions of the same functionality, greatly increasing development and code maintenance complexity.

Based on the analysis of performance results, it is possible to draw the following conclusions that can be used as guidelines for the geosimulation of multi-agent systems:

- It is important to choose the optimal simulation method depending on the number of agents and the complexity of the model. For simple models with a small number of agents, the overhead of the parallel simulation may turn out to be greater than the benefit. With an increase of parallel operation, overheads tend to be masked out and become less noticeable.
- It is crucial to choose the most appropriate data types for the model. Too low precision will have a significant impact on the accuracy of simulation results, while using high-precision types can greatly affect the simulation performance without giving additional benefits to simulation results.
- The technical differences between the arithmetic modules should be taken into account when comparing results between different platforms – the results obtained with different equipment may vary even in the case of identical models and it may be difficult to determine which of the results is more accurate.
- The simulation time may depend not so much on the total number of agents as on the density of the agents in the model. This factor must be considered when determining an optimal simulation method for long-running models by running a limited number of steps and comparing execution times – during simulation, due to changes of density, the chosen method might become inefficient.
- Sequential processing on GPU cores must be avoided whenever possible. It is also extremely important to always take into account the differences in sequential and parallel algorithms and to avoid assumptions that during the parallel execution a simulator will behave exactly the same as during sequential execution, even if the model itself will.
- Correct spatial index size, type and implementation must be chosen – inefficient spatial index might even lead to lower performance rather than its absence.
- It is recommended to avoid memory copying between CPU and GPU as this operation is relatively slow, but possible functional benefits when using copying must also be considered.

References

[1] "Parallel Programming and Computing Platform, CUDA" [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[2] "OpenCL - The open standard for parallel programming of heterogeneous systems" [Online]. Available: https://www.khronos.org/opencl

[3] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, "Overview and comparison of OpenCL and CUDA technology for GPGPU," 2012 *IEEE Asia Pacific Conference on Circuits and Systems*, Dec. 2012. https://doi.org/10.1109/apccas.2012.6419068

[4] "PTX ISA: Parallel Thread Execution ISA Version 5.0," 2017. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution

[5] I. Benenson and P. M. Torrens, *Geosimulation: Automata-based modelling of Urban Phenomena*. Chichester: John Wiley & Sons Ltd, 2004.

[6] M. Smith, P. Longley and M. Goodchild, "Geospatial Analysis: A comprehensive guide," 2015. [Online]. Available: http://www.spatialanalysisonline.com

[7] W. E. Easterling and K. Kok, "Emergent Properties of Scale in Global Environmental Modeling – Are There Any?," Integrated Assessment, vol. 3, no. 2–3, pp. 233–246, Jun. 2002. https://doi.org/10.1076/iaij.3.2.233.13576

[8] R. L. Goldstone and M. A. Janssen, "Computational models of collective behavior," *Trends in Cognitive Sciences*, vol. 9, no. 9, pp. 424–430, Sep. 2005. https://doi.org/10.1016/j.tics.2005.07.009

[9] M. Lysenko and R. D'Souza, "A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units," *Journal of Artificial Societies and Social Simulation*, vol. 11, no. 4, p. 10, 2008.

[10] "PEDSIM – A Pedestrian Crowd Simulation" [Online]. Available: http://PEDSIM.silmaril.org

[11] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics," *Physical Review E*, vol. 51, no. 5, pp. 4282–4286, May 1995. https://doi.org/10.1103/physreve.51.4282

[12] M. Moussaid, D. Helbing, S. Garnier, A. Johansson, M. Combe and G. Theraulaz, "Experimental study of the behavioural mechanisms underlying self-organization in human crowds," *Proceedings of the Royal Society B: Biological Sciences*, vol. 276, no. 1668, pp. 2755–2762, May 2009. https://doi.org/10.1098/rspb.2009.0405

[13] D. P. Ames, K. Asch, N. Bartelme, M. Becker and E. Al, *Springer Handbook of Geographic Information*, 1st ed. Würzburg: Springer-Verlag Berlin Heidelberg, 2012.

[14] R. Ding, X. Meng and Y. Bai, "Efficient index update for moving objects with future trajectories," in *Proceedings – 8th International Conference on Database Systems for Advanced Applications, DASFAA 2003*, 2003, pp. 183–191. https://doi.org/10.1109/dasfaa.2003.1192382

[15] "IEEE Standard for Floating-Point Arithmetic" [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

Jānis Cjoskāns graduated from Riga Technical University in 2017 with a Master degree in Information Technology. At present, he is the Head of Information Technology Division at Rural Support Service of the Republic of Latvia. His main areas of research include process anomaly detection and adaptive closed-loop process control algorithms and performance assessment.
E-mail: janis.cjoskans@lad.gov.lv

Arnis Lektauers, *Dr. sc. ing.*, is an Associate Professor at the Department of Modelling and Simulation of Riga Technical University (RTU). He has 16 years of professional experience delivering undergraduate and graduate courses at RTU, as well as developing more than 20 industrial and management information systems. His main scientific interests include the development of high performance interactive hybrid modelling and simulation solutions with the application of complex systems analysis and the research of industrial, economic, ecological and sustainable development problems. A. Lektauers is a member of the Latvian Simulation Society; the author of 1 textbook and more than 45 papers in scientific journals and conference proceedings in the field of information technology.
E-mail: arnis.lektauers@rtu.lv