Computing and Informatics, Vol. 34, 2015, 1418–1434

# GPGPU COMPUTING FOR MICROSCOPIC SIMULATIONS OF CROWD DYNAMICS

Jarosław Wąs, Hubert Mróz

*AGH University of Science and Technology*
*Faculty of Electrical Engineering, Automatics*
*Computer Science and Biomedical Engineering*
*al. Mickiewicza 30, 30-059 Kraków, Poland*
*e-mail:* jarek@agh.edu.pl

Paweł Topa

*AGH University of Science and Technology*
*Faculty of Computer Science, Electronics and Telecommunications*
*al. Mickiewicza 30, 30-059 Kraków, Poland*
*&*
*Institute of Geological Sciences, Biogeosystem Modelling Laboratory*
*Polish Academy of Sciences, Cracow Research Centre*
*Senacka 1, 31-002 Kraków, Poland*
*e-mail:* topa@agh.edu.pl

**Abstract.** We compare GPGPU implementations of two popular models of crowd dynamics. Specifically, we consider a continuous social force model, based on differential equations (molecular dynamics) and a discrete social distances model based on non-homogeneous cellular automata. For comparative purposes both models have been implemented in two versions: on the one hand using GPGPU technology, on the other hand using CPU only. We compare some significant characteristics of each model, for example: performance, memory consumption and issues of visualization. We also propose and test some possibilities for tuning the proposed algorithms for efficient GPU computations.

**Keywords:** Crowd simulation, social force, social distances, cellular automata, using GPU in simulation

## 1 INTRODUCTION

Effective and reliable modeling of crowd dynamics is currently an important issue. The results of such simulations are used by safety managers, fire engineers and security forces. More and more frequently the results of crowd simulation are used in order to test different scenarios (for instance dangerous situations [1]) during public gatherings. To improve the quality of simulations the concept of data-driven simulation is used, where actual data/attributes of pedestrians gained from video recordings or different electronic devices are continuously transferred to an on-line simulation.

It should be stressed, that currently in the vast majority of crowd simulators algorithms dedicated for engineering purposes that apply only the central processing unit (CPU) in computations are preferred, due to the fact that applying graphics processing units (GPUs) requires completely different algorithms and entire architectures, and such applications require rigorous tests. On the one hand, performing general purpose computing on graphics processor units (GPGPU) allows obtaining performance gains of several orders of magnitude compared with traditional CPU implementations in some cases. On the other hand, it is often impossible to reproduce complex rules/algorithms using GPU. This is because processing in GPU is realized by hundreds of processing units (e.g. CUDA Cores). The cores are grouped into streaming multiprocessors. Cores within a single streaming multiprocessor are commonly managed by a scheduler. The scheduler assigns to parallel execution threads grouped into a warp. Full performance of computation is achieved when all threads within the warp execute the same instructions, at the same time. This optimal situation from a computational point of view can be destroyed by branch instructions that might force some threads to process different instruction path. In such a case, the execution of thread is serialized, which heavily affects the performance.

In this article, we compare two approaches popular in crowd modeling, dedicated for engineering purposes, namely the social force model (based on differential equations) and a cellular automata based model – the social distances model. Two proposed models are connected with a microscopic – agent-based – simulation of a crowd. In such an approach each pedestrian is represented as a mobile entity. The aim of the research is to assess GPU technology for implementing continuous and discrete crowd models, to consider the applicability of GPU in reliable crowd simulators. The article is a continuation of two previously published articles by the authors, where the basic ideas of GPGPU application were presented [2, 3].

The application was created using NVidia CUDA (Compute Unified Device Architecture) technology. At the same time, 3D Object-oriented Graphics Rendering Engine (OGRE 3D) environment was used for the visualization.

The paper is structured as follows: Some related works are presented in Section 2. Next, some issues concerning using GPU in simulations are described in Section 3, whilst the issue of adapting algorithms for GPU and our two applied

models, continuous and discrete, are presented in Section 4. Consequently, the results of simulations are described in Section 5 and concluding remarks are placed in Section 6.

## 2 RELATED WORKS

The issue of the use of the GPU in modeling and simulation is becoming more and more important in recent years. One can observe a rapid growth in applications effectively using GPU in calculations, e.g. [4, 5, 6, 7]. Much theoretical and practical research is carried out in this field, for instance Bakhoda et al. [4] characterized several non-graphics applications written in NVIDIA's CUDA programming model and compared them against a CPU-only version of the application. Sunpyo and Hyesoon [9] proposed an analytic model that estimates the execution time of massively parallel programs using GPU.

We can find several interesting attempts at using GPU in crowd dynamics simulations. It should be stressed that such an approach is especially popular in simulations not created for engineering purposes.

Demeulemeester et al. proposed using a GPU during recalculations of potential fields in crowd simulations dedicated to computer games [10]. They developed simulation using the concepts of persistent threads and inter-block communication in GPU calculations. Similarly, Pelechano et al. [8] proposed an agent-based system of crowd dynamics based on differential equations and in further research they proposed using GPU for automatic generation of a navigation mesh.

Passos et al. [11] proposed using a fine-grained grid and accompanying data manipulation in order to lead to scalable algorithmic complexity in massive crowd simulations. They proposed such an implementation for game purposes of flocking boids, from which they ran benchmarks with more than one million simulated and rendered boids at nearly 30 fps.

Richmond et al. [12] presented a parallel framework for agent based modelling (ABM) exploiting the parallel architecture of the graphics processing unit (GPU). The framework included agent communication through messages with efficient use of shared memory. The publication was a part of FLexible Agent Modelling Environment (FLAME) project, devoted to agent-based simulations of complex systems.

An other use of the GPU cards in pedestrian dynamics was demonstrated by Machida and Naito in [13], where a pedestrian and vehicle detection framework was presented. The framework enables real-time processing of images obtained from a camera installed on a vehicle. They proposed a sliding-window – the cascade approach with multi-classifiers used for both the direction of a pedestrian and the distance of the pedestrian from a camera.

## 3 BASIC CONCEPTS OF GPU COMPUTATIONS

Graphic processor unit (GPU) was developed for processing graphic data that has a unique structure and features: regularity and weak dependencies between them. It is a perfect target for massive parallel processing. Whilst graphics systems allow only a fixed pipeline (e.g. OpenGL lower than 2.0), GPU was not an interesting tool for computational science. The situation changed when graphics vendors provided processors with programmable shader units. The feature allows a programmer to manipulate the data transferred to the processor.

Initially, the only programmer interfaces were shading languages such as Cg or GLSL. They were designed for graphic purposes, but if the problem was encoded into graphic data structures (e.g. textures), it was possible to use the processor for scientific computation (i.e. [14, 15]). Soon the main GPU vendors provided more flexible programming interfaces that allow using GPU to solve general purpose problems (GPGPU). The most advanced and most popular API is CUDA (Common Unified Device Architecture) developed by Nvidia Corporation and compatible only with Nvidia processors [16]. A competitive programming environment was proposed by IT companies associated with the Khronos Group [17]. OpenCL (Open Computing Language) was designed as an open programming framework that is able to utilize all computational resources [18]. Unlike CUDA, OpenCL has support from all processor vendors (Nvidia, AMD, Intel). For obvious reasons, OpenCL's support for Nvidia processors is not as good as in the case of CUDA.

Both CUDA and OpenCL are low-level programming interfaces which require experience and deep knowledge about GPU architecture from developers. For those who are not interested in mastering in GPU features possible good choices are OpenACC and OpenMP. These are the sets of compiler directives that allow easy parallel programming at thread level. OpenACC was intentionally developed to provide convenient support for GPU programming [19]. In its recent version, version 4.0, OpenMP offers support for various computing devices, including GPU [20].

The popularity of the CUDA programming environment is a result of intensive development of hardware and software. For Nvidia processors, a set of features supported by a given generation of processors is called Compute Capability (currently CC 3.5 in Kepler architecture). Development of hardware results in more powerful processors with higher number of computational cores, fast memory and advanced control units. Development of software provides programmers new instructions, data structures and functionality that allows more efficient programming. Nvidia also provides a complete programming environment, compiler, profiler and other tools that support the development of GPU-aware software.

Designing and implementing computationally efficient GPU algorithms is still a challenge [21]. It requires a very careful consideration of how the components of this architecture affect computation. In practice it means designing new algorithms and data structures.

## 4 IMPLEMENTATION OF CONTINUOUS AND DISCRETE CROWD MODELS USING GPU

### 4.1 Social Distances Model Implemented for GPU Computation

Cellular automata based simulations of crowd dynamics have become more and more popular in recent years. It is a popular approach in modeling and simulation of pedestrians [23, 24, 25], due to its effectiveness, massive parallelism and reliability.

The framework of cellular automata is often combined with the concept of potential fields [22]. In these models a set of static and dynamic fields is applied, and the fields modify the transition function of applied cellular automaton.

We also develop models of pedestrian dynamics based on cellular automata [29]. For instance, the social distances model was recently adapted for the mass evacuation of pedestrians [28].

In order to implement such a discrete model we have used two kinds of grids:

**Grid of occupancy** – the grid stores information about the appearance and position of an agent. This is the grid of a cellular automaton, when the state of particular cell is stored. A cell can be empty or occupied by an agent. An agent is represented as an ellipse, the center of which coincides with the center of a square cell. According to [28] the orientation of an agent is a state from the following set; $S_i \in \{Horizontal, Vertical, Right, Left\}$, which refers to an elliptical representation of pedestrian [29].

**Static potential field** – the grid stores information about the distance from a previously defined target (POI). A target is a source of a potential field, which is propagated to the next cell of a Moore neighborhood.

An agent can change the potential fields, but in a particular time slice he/she can be associated with only one potential field. The parallelism of calculations in this model is based on the simultaneous processing of multiple agents (one thread is associated with a single agent). The thread retrieves information from the neighborhood, it decides on the action and, if necessary, updates the information about the grid occupancy.

### 4.2 Social Force Model Implemented Using GPU

The social force model does not use any grid that establishes a static neighbourhood schema [30]. The space that contains moving agents is divided into containers. We use containers sized $15\,\mathrm{m} \times 15\,\mathrm{m}$. It should be emphasized that during calculations we have taken into account only forces associated with a current container. Each container is assigned to different threads, which execute all steps necessary to update the agent position, direction and speed. We apply the following steps in our social force algorithms:

- calculate force affecting agent
- resolve the collisions
- update the position of agent
- update the speed of agent
- update the direction of agent movement

This model of parallelism is relatively difficult to implement in GPU. The main problem is the different number of agents in each container, especially after several steps of simulations, when agents are concentrated around their targets. Additionally, moving agents can leave containers due to their dynamics and defined aims. These situations cannot be directly handled by kernels because it requires synchronization and communication between them, when agents are exchanged between containers. In our approach, after a single step of computation, data is collected and sent to the host, where the contents of containers are updated.

### 4.3 Calculation Scheme

We have used the scheme of calculations presented in Figure 1. In our approach *Simulation manager* is the master unit, which watches over the course of the simulation. Next, the chosen *Simulation model* is initialized, as well as *Calculation Module* and memory is allocated. Afterwards, the simulation is performed, and all information about the agents is actualized iteratively.

### 4.4 Allocation of Memory in Discrete and Continuous Crowd Models

Among most important aspects of the models developed for large simulations are the issues of memory occupancy. This is particularly important in applications using GPUs (amount of available memory). The implementation of a discrete model, as well as a continuous one, in terms of used memory, was determined by two variables: the number of implemented agents and the size of the simulation world. Other objects like buildings or obstacles have a marginal impact on memory.

As we have mentioned above, implementation of the social force model is based on the idea of containers. An important factor influencing the performance of the application is the selection of their optimal size. Why? The main reason for looking for optimal container size is the fact, that having a large number of small containers causes high memory consumption, whilst the introduction of large containers will reduce the speed of the simulation, because the number of agents located in a particular container increases, and in such a situation it is necessary to perform calculations between each of them. As a result of our tests, the decision was made to determine the size of the container to be 15 meters. This means that each container requires 276 bytes of data to store.

In the social distances model we have identified those elements with the greatest impact on the memory occupation as applied grids. In the minimal case two grids

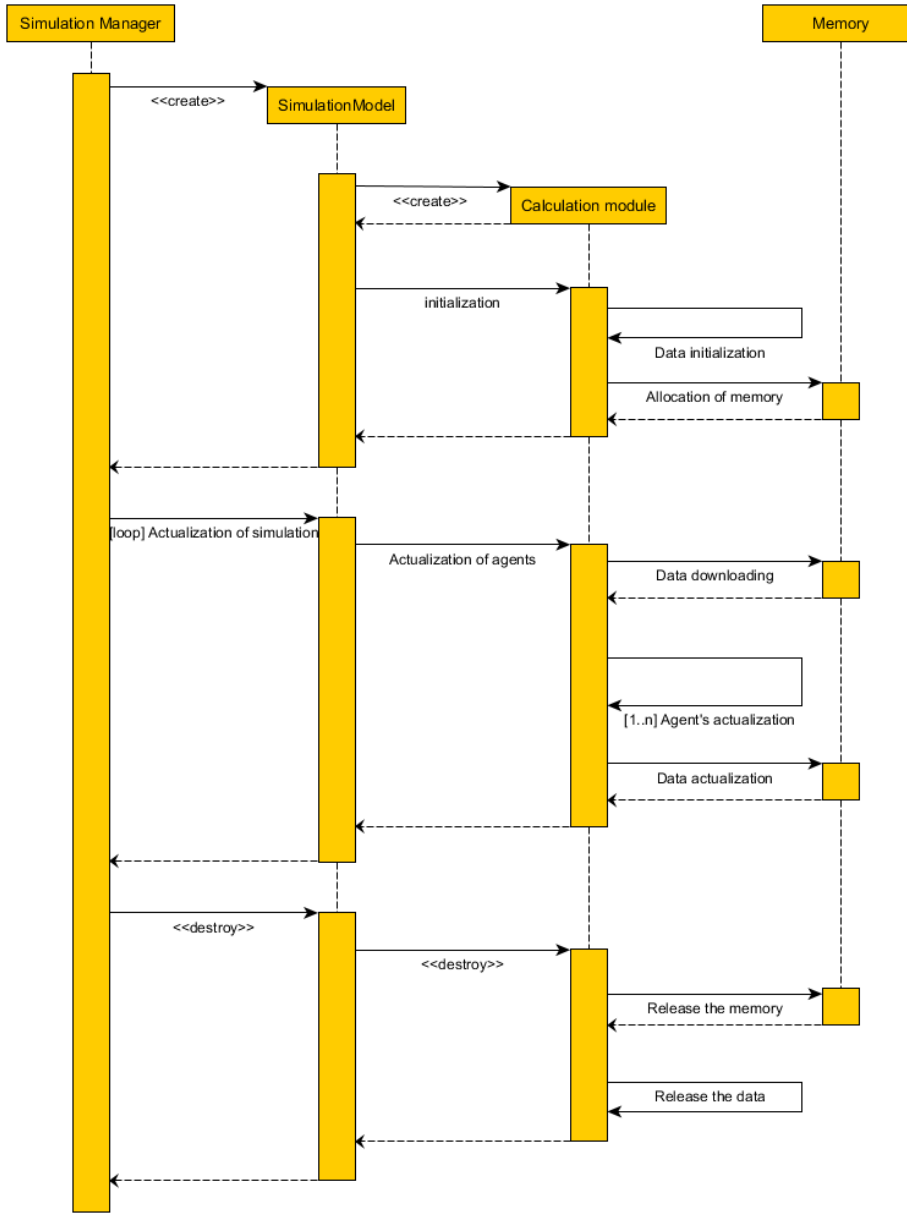


Figure 1. Sequence diagram of presented application

are created: a potential grid (static potential field) and an occupancy grid. In our model the size of each cell is 0.25 meters. One cell of grid requires 5 bytes of memory (2 bytes for the occupancy grid and 3 bytes for potential field grid). This gives 300 bytes for the 15-meter area in the social distances model compared 276 bytes in the social force model. However, each additional grid in the CA-based model increases this value by 180 bytes. This does not constitute a significant limitation in the simulation of evacuation, but in the case of free movement (with many possible targets) this should be perceived as a limitation.

We present levels of memory occupancy according to implemented number of agents (see Table 1), and according to simulation world size (see Table 2).

| Number of Agents | Social Force Model | Social Distances Model |
|---|---|---|
| 1 000 | 0.031 MB | 0.08 MB |
| 5 000 | 0.153 MB | 0.038 MB |
| 10 000 | 0.305 MB | 0.076 MB |
| 50 000 | 1.526 MB | 0.381 MB |
| 100 000 | 3.052 MB | 0.763 MB |
| 500 000 | 15.259 MB | 3.815 MB |
| 1 000 000 | 30.518 MB | 7.629 MB |

Table 1. Data size of the array of agents (with rendering disabled)

| World Size | Social Force | Social Distances – 2 grids | Social Distances – 5 grids |
|---|---|---|---|
| 50 m | 0.004 MB | 0.114 MB | 0.420 MB |
| 100 m | 0.013 MB | 0.458 MB | 1.678 MB |
| 500 m | 0.304 MB | 11.444 MB | 41.962 MB |
| 1 000 m | 1.182 MB | 45.776 MB | 167.847 MB |
| 2 500 m | 7.34 MB | 286.102 MB | 1 049.042 MB |
| 5 000 m | 29.363 MB | 1 144.409 MB | 4 196.168 MB |

Table 2. Data size of the simulation world (with rendering disabled)

## 5 SIMULATION RESULTS

### 5.1 Level of Details

In order to implement our project, we have used Nvidia CUDA programming platforms. The applied technology enables the creation of executable code on CPU, as well as on GPU. We have also compared the differences in performance. Tests presented in the paper were carried out on a hardware platform equipped with: Dual-Core AMD Athlon II 250 (3.00 GHz), 4 GB of RAM and a GeForce GTS 250 graphics card with 512 MB RAM and Compute Capability 1.1.

An object-oriented Graphics Rendering Engine was used for rendering the graphical part of the simulation as performed additional task. Three scenarios were used

for handling graphics: full rendering, simplified rendering (Level of Details) and no rendering. We present screen-shots of our application for full rendering mode (Figure 2 on the left) and for simplified rendering – LoD (Figure 2 on the right).

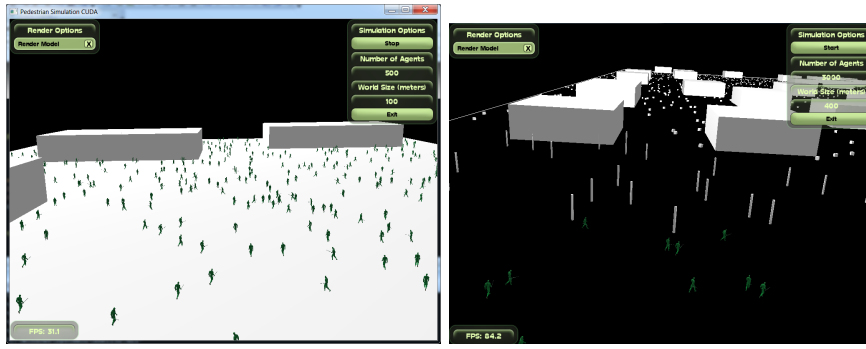


Figure 2. Two modes of visualization in our application – full rendering and simplified rendering (Level of Details)

Table 3 contains rendering performance for 500 acting agents and environment size of 100 m × 100 m, while Table 4 contains rendering performance for 2 500 agents and environment size of 250 m × 250 m.

Unfortunately the rendering system appears to be insufficient in large agent populations. In the future we would like to implement a dedicated rendering system that retrieves necessary information directly from the existing data blocks in the device memory.

| Model | Performance Full Rendering | Performance Level of Details | Performance Simplified Rendering |
|---|---|---|---|
| Social Force | 29 FPS | 78 FPS | 98 FPS |
| Social Distances | 35 FPS | 100 FPS | 122 FPS |

Table 3. Performance of rendering of 500 agents, environment size equals 100 m × 100 m

| Model | Performance Full Rendering | Performance Level of Details | Performance Simplified Rendering |
|---|---|---|---|
| Social Force | 8 FPS | 25 FPS | 28 FPS |
| Social Distances | 9 FPS | 32 FPS | 39 FPS |

Table 4. Performance of rendering of 2 500 agents, environment size equals 250 m × 250 m

### 5.2 Usage of Grids of Potential Fields

For more complex environments we can define many targets and for each class of target we have to define a different, static potential field. An agent heading to

a target is assigned to one occupancy grid, or one of the available potential grids (static potential field).

We have also tested the impact of the number of potential grids (potential fields) on performance of the simulation. The tests were performed several times, for a population of 5 000 agents, in an area of 500 × 500 meters (Table 5). Rendering of three-dimensional objects has been disabled.

| | 1 grid | 5 grids | 10 grids | 15 grids | 20 grids |
|---|---|---|---|---|---|
| CPU | 178 FPS | 176 FPS | 173 FPS | 168 FPS | 165 FPS |
| GPU | 295 FPS | 298 FPS | 299 FPS | 295 FPS | 293 FPS |

Table 5. Performance tests for different numbers of potential fields

As the results show (Table 5), increasing the number of grids does not affect significantly the performance of the simulation. Small differences result mainly from the random generation of the simulation world e.g. agent positions, buildings and targets, which can affect performance. No greater influence on the number of grids (on simulation performance) is a very important observation. This lets us know that, while ensuring sufficient space in the memory, we are able to carry out complex simulations (using a large number of grids) without affecting its performance.

### 5.3 Towards Efficient GPU Computation

A typical program applying GPU for computation consist of two parts: the first one is executed by a host processor and the second one is issued to run on a graphic processor. The main task of the host-executed code is to prepare data for computation and call GPU-executed functions called kernels (part). Moreover, in the case of more complex problems, the part of the task that cannot be parallelized is also executed by the host.

In social force and cellular automata approach (namely the social distances model) we also use the same schema of task partitioning. The host code prepares and initializes data for model and sends them to the global memory of the GPU device. In each step of simulation the kernel that updates position of agents is called. In both models, the original algorithms are relatively complex with many branch instructions. Also, especially in the case of social force model, the date cannot be efficiently distributed among the threads.

In order to use streaming processors efficiently, we must ensure that the scheduler will be able to fill them by threads. This is only possible when all threads execute exactly the same instructions, thus branch instructions should be avoided. In this task the profiler supplied by Nvidia was especially helpful, identifying branch instructions that generate divergent executions.

In such a complex code where many new values are calculated at each step of simulation, eliminating problematic "if" instructions is not an easy task and in some situations it is impossible. Moreover, such modifications make the code unclear and

difficult to analyse. As an example, below we present, how "if" instructions can be replaced by an arithmetical calculation.

```
__device__ float angleBetween(const float2& v1,
                              const float2& v2)
{
        float angle = atan2(v2.y, v2.x) - atan2(v1.y, v1.x);
        if(angle >= PI)
                angle = 2 * PI - angle;
        if(angle <= -PI)
                angle = -2 * PI - angle;
        return angle;
}
```

This simple code is identified by a profiler as a reason for divergent executions. It can be replaced by the following code:

```
__device__ float step(float a, float x)
{
        return x>=a;
}

__device__ float angleBetween(const float2& v1,
                              const float2& v2)
{
        float i = step(PI, angle);
        angle = 2*PI * i - (2*i - 1)*angle;
        i = step(angle, -PI);
        angle = - 2*PI * i - (2*i-1)*angle;
        return angle;
}
```

Branch divergence occurs only for threads that are grouped in the same warp. Thus, in order to avoid branch divergence, the calculations should be organized in such a way that threads within a single warp execute the same sequence of instructions. Other instruction paths should be performed by a thread in another warp (see [27] for a more detailed explanation).

Memory transactions are another area where optimization should be performed. The highest performance and lowest latency are provided by registers. These are exclusively used by a single thread for storing its local data. The small capacity of the registers and their limited number is another bottleneck in this architecture. When the number of local variables used within a single thread exceeds the limit of registers, they can be stored in slower shared/local memory (register spilling). Another issue is the total amount of registers that are distributed across the threads. If a single thread uses a higher number of registers, a lower number of threads can be executed at the same time. The original algorithms in both models use a relatively

large number of registers (32 for social distances model and 48 for social force model), which significantly affects occupancy parameter.

The use of fast shared memory (local memory) ca be considered another area of optimization. It has almost the same speed and latency as registers. It is located on-chip, thus its capacity is also relatively small (up to 48 KB in Kepler processors). All threads in the same block of threads have access to the same shared memory. Access to shared memory has to be carefully organized in order to avoid bank conflicts which destroy parallel transactions [27]. In [26] the potential effects of using shared memory were concluded to be:

- for GPU with CC $<$ 2.0 shared memory gives speedup 1.5 to 4 times faster compared to algorithms that use only global memory
- for GPU with CC $>=$ 2.0 algorithms that use shared memory are as good as algorithms that use only global memory

This effect comes from the fact that in case of a GPU with CC $>=$ 2.0 (Fermi and Kepler architecture) global transactions are cached. In our opinion, in this area optimization should concentrate on ensuring that transactions to/from global memory are coalesced into larger blocks (64 or 126 bits).

### 5.4 Performance Tests

In order to verify the efficiency of the presented models, a series of tests have been carried out. Criterion used for comparison purposes was the speed of the simulation, expressed in frames per second (related to number of agents). Computational effort and time required to perform the calculations in the simulation of crowd dynamics is closely related to the density of the population of agents. Therefore, tests were carried out in a similar population density for continuous and discrete models using CPU and GPU technologies.

The tests described in the current section were performed with rendering disabled (because of the inefficient rendering module).

The chart in Figure 3 shows the results of performance tests for social force and social distances models carried out respectively on CPU and GPU. Performance in this case is expressed by the relationship: number of supported pedestrians to FPS (frame per second). As we can see on the chart for both models, simulations executed on GPU are characterized by a higher performance, than the corresponding CPU simulations. It is worthwhile to emphasize that this increase is achieved for non-optimized algorithms.

As a discrete model, social distances in tests showed much greater efficiency than continuous social force: on CPU results are better by 160–300 %. By using GPU we are able to improve results by an additional 60–70 %. As we mentioned earlier, we still have space for improvements by optimizing GPU algorithms. The increase of the performance for the discrete model was lower for GPU, because it requires a large amount of memory references. This is due to the fact that threads

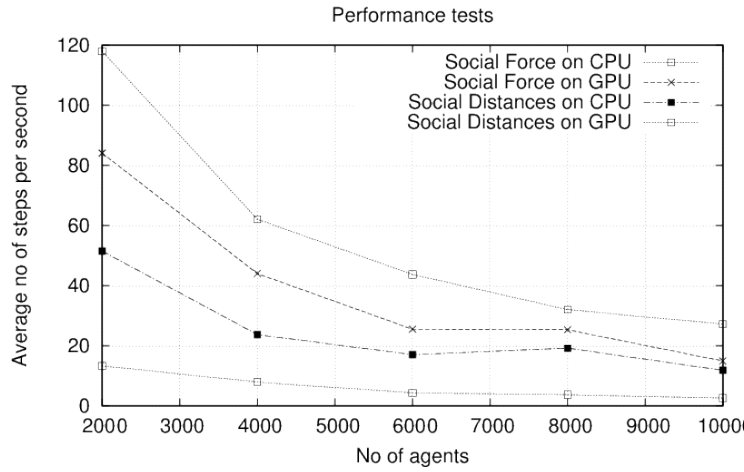


Figure 3. Results of performance tests for social force and social distances models implemented respectively on CPU and GPU

related to agents must (during operation) access information about their neighborhood, and then update the data in an occupancy field. This results in a memory overhead, because operations related to memory access are very expensive (in a calculation) and may lead to bottlenecks in GPU applications. However, further optimizations can be achieved in this field. CUDA allows users to take advantage of different types of memory, such as global, shared or texture memory. All of them have pros and cons and appropriate usage can be crucial in an application's performance.

## 6 SUMMARY

GPU technology has a great potential in the field of simulation acceleration, and requires a different approach to creating a simulation framework and particular algorithms when compared to the "traditional" approach.

The aim of our article was to apply GPGPU computations in microscopic crowd simulation and to compare two popular microscopic models of crowd dynamics. The first one is the differential equations based continuous social force model, whilst the second one is the cellular automata based discrete social distances model.

Both models have been implemented using GPGPU and, for comparative purposes, using CPU. It should be noted, that the use of GPU requires a completely different approach to the development of applications and in this case many limitations will appear. This issue was addressed in Section 3. It should be noted, that

for technical reasons connected with using GPU (limitations in creating of complex instructions, inadvisable data transfer etc.) final algorithms should be simplified in comparison to the latest implementations of crowd dynamics models by the authors [28, 31] (significantly simplified transition function, limited reproduction of operational, tactical and strategic level of decision-making, etc.).

Based on the analysis of many variants of size and environment complexity and the available number of agents, the authors proposed ranges of applicability for the two models created using the GPU technology. This was addressed in Section 5.

The social force model works well in large, sparsely populated areas, where the density of agents is relatively small. This continuous model works perfectly for free pedestrian traffic, where agents head for a large number of defined targets.

The discrete social distances model gives good results for the simulation of objects with a large number of obstacles and a large population of agents. Due to the necessity of application in many potential fields, the GPU implementation of social distances model is suitable for simulations in which the number of pedestrian targets is not high: it can be applied in evacuation scenarios or in relatively simple, freeway traffic scenarios. The social force method is continuous, thus it allows more accurate mapping of pedestrian movement. The trajectories of motion are much more accurate than in a discrete model. On the other hand, the social distances method offers the possibility to generation an environment with more complex topology (building, obstacle), whilst in social force model it is more problematic.

Currently, the development of GPU technology is rapid and the authors expect that GPGPU will be more and more competitive when compared to traditional CPU programming. During the implementation of the models presented in this work, the specification of CUDA technology has been changed several times and some new features have been included (for example *Thrust* library of templated primitives). Currently, most professional applications dedicated to crowd dynamics, especially applications developed for engineering purposes (such as evacuation, crowd management, etc.), implement a simulation engine using traditional CPU. However, the immediate future may bring a change in this area.

We found some issues that should be addressed in the future. The first is the issue of a branch divergence parameter that should be lowered by eliminating conditional instructions or grouping threads executing the same path of instructions into the same warps. The second one is connected with more optimal usage of registers in order to enable more efficient SM (shared memory) utilization. The last issue is arranging data to achieve memory coalescing in store/load transactions.

## Acknowledgments

Jarosław Wąs and Hubert Mróz gratefully acknowledge that this research is partially supported by AGH University of Science and Technology, contract No. 11.11.120.859. Paweł Topa acknowledges the partial support from AGH University of Science and Technology, contract No. 11.11.230.124.

## REFERENCES


[1] Kryza, B.—Król, D.—Wrzeszcz, M.—Dutka, Ł.—Kitowski, J.: Interactive Cloud Data Farming Environment for Military Mission Planning Support. Computer Science, Vol. 13, 2012, No. 3, pp. 89–100.

[2] Mróz, H.—Wąs, J.: Discrete vs. Continuous Approach in Crowd Dynamics Modelling Using GPU Computing. Journal Cybernetics and Systems, Vol. 45, 2014, No. 1, pp. 25–38.

[3] Mróz, H.—Wąs, J.—Topa, P.: The Use of GPGPU in Continuous and Discrete Models of Crowd Dynamics. Proceedings of the 10th International Conference on Parallel Processing and Applied Mathematics (PPAM 2013), Warsaw, Poland, 2013, Revised Selected Papers, Part II, LNCS, Vol. 8385, 2014, pp. 679–688.

[4] Bakhoda, A.—Yuan, G.—Fung, W. W. L.—Wong, H.—Aamodt, T. M.: Analyzing CUDA Workloads Using a Detailed GPU Simulator. Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), 2009, pp. 163–174.

[5] Blecic, I.—Cecchini, A.—Trunfio, G. A.: Cellular Automata Simulation of Urban Dynamics through GPGPU. The Journal of Supercomputing, Vol. 65, 2013, No. 2, pp. 614–629.

[6] Yuan, F.: An Interactive Concave Volume Clipping Method Based of GPU Ray Casting with Boolean Operation. Computing and Informatics, Vol. 31, 2012, No. 3, pp. 551–571.

[7] Worecki, M.—Wcisło, R.: GPU Enhanced Simulation of Angiogenesis. Computer Science, Vol. 13, 2012, No. 1, pp. 35–48.

[8] Pelechano, N.—Allbeck, J. M.—Badler, N. I.: Controlling Individual Agents in High-Density Crowd Simulation. Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '07). Eurographics Association, Aire-la-Ville, Switzerland, 2007, pp. 99–108.

[9] Sunpyo, H.—Hyesoon, K.: An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. SIGARCH Computer Architecture News, Vol. 37, 2009, No. 3, pp. 152–163.

[10] Demeulemeester, A.—Hollemeersch, C. F.—Mees, P.—Pieters, B.—Lambert, P.—Van de Walle, R.: Hybrid Path Planning for Massive Crowd Simulation on the GPU. Motion in Games, LNCS, Vol. 7060, 2011, pp. 304–315.

[11] Passos, E. B.—Joselli, M.—Zamith, M.—Gonzalez-Clua, E. W.—Montenegro, A.—Conci, A.—Feijo, B.: A Bidimensional Data Structure and Spatial Optimization for Supermassive Crowd Simulation on GPU. ACM Computers in Entertainment (CIE), Vol. 7, 2009, No. 4, 15 pp., DOI: 10.1145/1658866.1658879.

[12] Richmond, P.—Coakley, S.—Romano, D. M.: A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA. Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09), 2009, Vol. 2, pp. 1125–1126.

[13] Machida, T.—Naito, T.: GPU and CPU Cooperative Accelerated Pedestrian and Vehicle Detection. 2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops), 2011, pp. 506–513.

[14] Gobron, S.—Devillard, F.—Heit, B.: Retina Simulation Using Cellular Automata and GPU Programming. Machine Vision and Application, Vol. 18, 2007, No. 6, pp. 331–342.

[15] Rumpf, M.—Strzodka, R.: Graphic Processor Units: new Prospects for Parallel Computing. Numerical Solution of Partial Differential Equations on Parallel Computers, Lecture Notes in Computational Science and Engineering, Vol. 51, 2006, pp. 89–132.

[16] Nvidia CUDA, `http://www.nvidia.com/object/cuda_home_new.html`.

[17] The Khronos Group, `http://www.khronos.org/`.

[18] OpenCL, The open standard for parallel programming of heterogeneous systems, `https://www.khronos.org/opencl/`.

[19] OpenACC, Directives for accelerators, `http://www.openacc-standard.org/`.

[20] OpenMP Application Program Interface, Version 4.0, `http://www.openmp.org`, July 2013.

[21] Mann, Z. Á.: GPGPU: Hardware/Software Co-Design for the Masses. Computing and Informatics, Vol. 30, 2011, pp. 1247–1257.

[22] Burstedde, C.—Klauck, K.—Schadschneider, A.—Zittartz, A.: Simulation of Pedestrian Dynamics Using a Two-Dimensional Cellular Automaton. Physica A, Vol. 295, 2001, No. 3-4, pp. 507–525.

[23] Zhang, P.—Jian, X. X.—Wong, S. C.—Choi, K.: Potential Field Cellular Automata Model for Pedestrian Flow. Physical Review E, Vol. 85, 2012, No. 2, Art. No. 021119.

[24] Koyama, S.—Shinozaki, N.—Morishita, S.: Pedestrian Flow Modeling Using Cellular Automata Based on the Japanese Public Guideline and Application to Evacuation Simulation. Journal of Cellular Automata, Vol. 8, 2013, No. 5-6, pp. 361–382.

[25] Dietrich, F.—Köster, G.—Seitz, M.—von Sivers, I.: Bridging the Gap: From Cellular Automata to Differential Equation Models for Pedestrian Dynamics. Journal of Computational Science, Vol. 5, 2014, No. 5, pp. 841–846.

[26] Topa, P.: Cellular Automata Model Tuned for Efficient Computation on GPU with Global Memory Cache. Proceedings of 2014 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Torino, Italy, 2014, pp. 380–383.

[27] Topa, P.—Młocek, P.: Using Shared Memory as a Cache in Cellular Automata Water Flow Simulations on GPUs. Computer Science, Vol. 14, 2013, No. 3, pp. 385–401.

[28] Wąs, J.—Lubas, R.: Adapting Social Distances Model for Mass Evacuation Simulation. Journal of Cellular Automata, Vol. 8, 2013, No. 5-6, pp. 395–405.

[29] Wąs, J.—Gudowski, B.—Matuszyk, P. J.: New Cellular Automata Model of Pedestrian Representation Incorporating Proxemics into People Dynamics. ACRI 2006, LNCS, Vol. 4173, 2006, pp. 724–727.

[30] Helbing, D.—Molnar, P.: Social Force Model for Pedestrian Dynamics. Phys. Rev. E, Vol. 51, 1995, No. 5, pp. 4282–4286.

[31] Wąs, J.—Lubas, R.: Towards Realistic and Effective Agent-Based Models of Crowd Dynamics. Neurocomputing, Vol. 146, 2014, Iss. C, pp. 199–209.

**Jarosław Wąs** is currently Assistant Professor at AGH University of Science and Technology in the Department of Applied Computer Science. After graduation in 1999 he worked in ComArch S.A. as a system analyst, next he successfully completed several projects in Boccard as a project manager. Afterwards, he worked as a junior researcher at AGH University (from 2001) in the Faculty of Electrical Engineering, Automatics, Computer Science and Electronics. In 2006 he finished his Ph.D. in the area of crowd dynamics modeling, at AGH University. His research areas are connected with agent-based modeling, cellular automata, complex systems and crowd dynamics. He has been responsible for large-scale crowd models in national and international projects.

**Hubert Mróz** studied applied computer science at the AGH University of Science and Technology in Cracow, in the Faculty of Electrical Engineering, Automatics, Computer Science and Electronics and in 2011 he completed his Master's degree. Since 2009 he is working in game industry as a programmer and designer. He was responsible for gameplay mechanics, multiplayer implementation, AI and tools, among other things. He took part in over a dozen of projects, released on most available game platforms (personal computers, handheld and stationary game consoles, mobile devices, web browsers). He also used to work in Bloober Team S.A. and iFun4All Sp. z o.o. He was one of the founders of Tap It Games company. He is also working on his own game projects.

**Paweł Topa** is currently Assistant Professor at AGH University of Science and Technology in the Department of Computer Science. Since his graduation in 1999 he is working at AGH University of Science and Technology. In 2005 he defended his doctoral thesis proposing new tools for modeling phenomena in the area of geology and micropalaeontology. Since 2010, he works also at Institute of Geological Sciences, Polish Academy of Sciences as Assistant Professor. His scientific interests cover cellular automata theory and applications, complex systems modelling, GPU computing.