Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200029

GPGPU Computing for Microscopic Pedestrian Simulation

Benedikt Zönnchen [a,b,1], Gerta Köster [a]
[a] *Munich University of Applied Sciences, Lothstrae 64, 80335 Munich, Germany*
[b] *Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany*

Abstract. GPGPU computation of microscopic pedestrian simulations has been largely restricted to Cellular Automata and differential equations models, leaving out most agent-based models that rely on sequential updates. We combine a linked-cell data structure to reduce neighborhood complexity with a massive parallel filtering technique to identify agents that can be updated in parallel, thus extending GPGPU computation to one such model, the Optimal Steps Model. We compare two different OpenCL implementations: a parallel event-driven update scheme and a parallel update scheme that violates the event order for the sake of parallelism. We achieve significant speed ups for both in two benchmark scenarios making faster than real-time simulations possible even for large-scale scenarios.

Keywords. discrete event simulation, agent-based simulation, pedestrian dynamics, GPGPU, linked cell algorithm

1. Introduction

Modelling of crowd dynamics has become an important area of research. It helps to understand the interaction of large crowds on a macroscopic level. Results of reliable crowd simulations support safety managers, engineers, event managers and security staff in their decisions. Off-line simulations allow testing of architectural solutions for buildings or facilities for mass events. Nowadays, the application of pedestrians simulations goes beyond off-line simulations. There is a growing interest in and a need for on-line data-driven simulations. Such simulations can predict the future — but only if the computation is faster than real-time. Since microscopic crowd simulations are computationally expensive, they must be accelerated to enable predictive simulations on a large scale.

With the breakdown of Dennard scaling, clock frequencies of single Central Processing Units (CPUs) no longer increase significantly. As a consequence, manufacturers turned their attention towards multi-core processors. In contrast to CPUs, the hardware architecture of Graphics Processing Units (GPU) is designed for massive parallelism. Since GPUs are part of many current and upcoming supercomputers, efficient exploitation of GPUs has become essential in scientific computing. Additionally, GPUs offer thousands of cores inside affordable off-the-shelf workstations making general-purpose Graphics Processing Units (GPGPUs) a source of cheap and efficient computational power. In crowd dynamics thousands or even millions of virtual pedestrians move

[1]Corresponding Author: Benedikt Zönnchen: *zoennchen.benendikt@hm.edu*

simultaneously. At the same time, they are spatially separated, which implies that there is a chance for parallel updates. Consequently, we consider how to exploit GPUs for large-scale crowd simulations.

To simulate thousands of pedestrians in real-time, Cellular Automata (CA) based models are an attractive choice. Space is discretized by a regular and fixed grid of cells and agents are usually represented by occupied cells. This regularity induces efficiency with respect to computational complexity even without parallelism but it also causes inflexibility and inaccuracy in terms of modelling. Motion is restricted to the grid making CA unsuitable for scenarios with high pedestrian densities or fine spatial granularity. (GPGPU) for CA modelling has been explored and successfully applied by many researchers [1,2,3,4,5,6].

Other wide-spread classes of microscopic pedestrian models are force- and velocity-based models where a set of ordinary differential equations (ODEs) describes motion. In contrast to Cellular Automata, agents move in continuous space. Discretization of the continuous model is necessary to numerically solve the equations. Especially for crowded scenarios, accurate results imply small time steps and thus a lot of computational power. In [3] the authors discuss GPU implementations of a CA model, the Social Distance Model (SDM), and the force-based Social Force Model (SFM). They achieve a speed up factor of approximately 4 by exploiting GPGPU.

Almost all microscopic models are, in fact, agent-based models (ABMs). There is a lot of research on using hardware accelerators for ABMs but mostly outside the field of pedestrian dynamics. An extensive overview can be found in [7].

In this contribution we extend massive parallelism through GPUs to another class of agent-based pedestrian models represented by the well validated Optimal Steps Model (OSM) [8,9,10]. In the OSM each agent steps ahead in two dimensional space, driven by its individual pace. The agent's next position is found by solving an optimization problem. Thus the OSM is discrete in time and continuous in space. We present and compare two implementations of the OSM which differ in their update schemes: an inherently sequential event-driven update scheme, which is the OSM's original update scheme, and a newer parallel update scheme.

2. The Optimal Steps Model

The OSM combines aspects from both, CA and differential equation models. It inherits rule-based discrete stepping events from CA and motion in continuous space from differential equation models. Furthermore, it can be classified as an agent-based model (ABM), since each agent is individualized by its unique free-flow speed v_{free} and stride length λ. The principle idea behind the OSM is that the natural stepwise movement of pedestrians leads to a spatial discretization within the simulation [11,9]: Let Ω be the simulated spatial domain and $\Omega_{\text{out}} \subset \Omega$ the obstacle domain, that is, all space covered by obstacles such as walls. Let $\Omega_{\text{in}} = \Omega \setminus \Omega_{\text{out}}$ be the walkable part of the scenario. Furthermore, let $\partial\Omega_{\text{out}}$ be the scenario boundary. Pedestrians are represented by circular shaped agents of radius $r_p = 0.195$ meters. They move inside Ω_{in}. Agents can step forward in any direction by choosing a position inside their stepping circle. See Fig. 1a. The radius of an agent's stepping circle is derived from the experimentally observed linear dependency of the stride length on the free-flow speed presented in [8]. That is, the radius is given by

$$\lambda = \beta_0 + \beta_1 \times v_{\text{free}} + \varepsilon, \tag{1}$$

where v_{free} is the agent's free-flow speed and ε is a normally distributed error term, $\varepsilon \sim \mathcal{N}(0,\sigma)$. The intercept β_0 and slope β_1 stem from a regression through experimental data [8]. Therefore, λ represents the natural stride length of the the modelled pedestrian. To obtain a heterogeneous population, the free-flow speed is chosen from a truncated normal distribution. Let x_k be the current position of agent i and τ_i be the event time of its next step, then the next position x_{k+1} is found by optimizing a utility function Φ within the stepping circle around the agent:

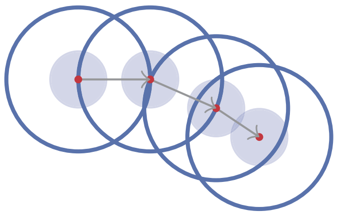
$$x_{k+1} = \arg\min_{y\,\in\,P_i} \Phi_i(y), \quad \text{with} \quad P_i = \{y : \|y - x_k\| \leq \lambda_i\} \tag{2}$$

The Optimal Steps Model is event driven. In fact, the linear dependency between free-flow speed and step length also uniquely determines each agent's pace: While the model moves the agent to x_{k+1} in an instant, its next footstep event occurs at $\tau_i + \lambda_i / v_{i,\text{free}}$. In our free and open implementation of the OSM [10], the optimization problem is currently solved either by the Nelder-Mead method or by a brute force evaluation on a discretization of P_i [12] visualized in Fig. 1.

The utility function Φ, which is often interpreted as a potential field, ensures that the destination is reached while skirting obstacles and other agents. We consider it more closely, because calculating Φ contains computationally expensive steps. Let Φ_i be the utility function, or potential field, of agent i. It is given by a sum of sub-utilities or sub-potentials: $\Phi_i = \Phi_{t,\Gamma} + \Phi_o + \Phi_{p,i}$.

- $\Phi_{t,\Gamma}$: contributes attraction to a target Γ and is given by the solution of the eikonal equation. $\Phi_{t,\Gamma}(x)$ encodes the travel time required to reach Γ starting from x. All agents approaching the same target share a common target potential field.
- Φ_o: contributes repulsion caused by obstacles and depends on the distance $d_\Omega(x) = \min_{y \in \partial\Omega} \|x - y\|$, which is the shortest distance to the closest obstacle.
- $\Phi_{p,i}$: is the sum of local repulsion terms caused by other agents and depends on the distance to these agents.

The target and obstacle potential fields are static but Φ_p changes dynamically with the movement of agents. Both repulsive potentials increase with decreasing distance to ob-


a) Ilustration of footsteps of agent i. The circles (blue) indicate the actual step radius λ_i and the shaded area (blue) represents the agent torso of radius r_p.

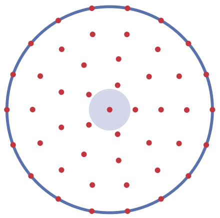

b) Approximation of P_i by equidistant points inside the step circle.

Figure 1. Computation of the next agent position.

stacles and other agents, respectively. Each sub-potential of Φ_p is realized by a function that has compact support, that is, it is zero outside a small area of influence. For a more detailed description of the modelling aspects we refer to [11,9].

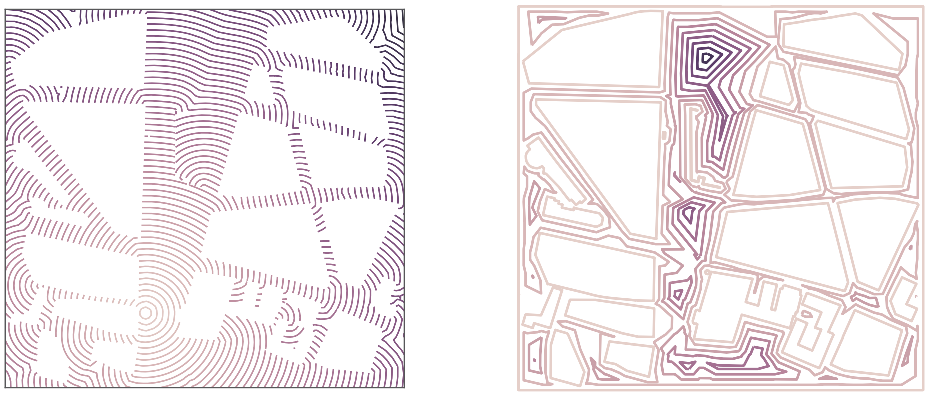
Regarding computational complexity Φ_p is the crucial part. Each sub-potential includes the evaluation of a square root and an exponential function. Therefore, we aim at computing as many sub-potentials as possible in parallel. One essential property to work with is that Φ_p is a local function. More precisely, if x is the position of agent j with $j \neq i$ then its contribution to $\Phi_{p,i}$ at y is zero if and only if $\|x - y\| > w_p$. The locality property and the fact that agents are spatially separated imply that footstep events of agent are not likely to interfere with each other if they are close in time but distant in space. This permits us to exploit parallelism.

2.1. The Event-driven Update Scheme

The orginal OSM is event-driven. The event-driven update scheme processes events in their natural order, that is, the way they occur. In terms of the OSM this ensures that for the choice of the next footstep at time t all footstep events which starts at $\tau < t$ are already processed. From a modelling perspective this implies that pedestrians can anticipate currently processed footsteps of nearby pedestrians. Therefore, Φ_p actually depends on agents' positions in the very near future. By using the event-driven update scheme the OSM becomes a discrete event simulation (DES) model. Note that even though pedestrians only anticipate the movement of nearby pedestrians, this can lead to a chain of navigation adaptations propagating through the whole spatial domain.

2.2. The Parallel Update Scheme

An alternative implementation of the OSM presented in [13] suggest a parallel update scheme. The parallel update scheme uses a global synchronizing clock. An increase of the clock by a fixed time step Δt processes all footstep events within $(t; t + \Delta t]$ in parallel.



a) A target potential $\hat{\Phi}_{t,\Gamma}$ spreads out like wave from a target on the bottom left.

b) Distance function $\hat{d}_\Omega$ which gives the minimal distance to the nearest obstacle.

Figure 2. Plot of solutions of the Eikonal equation of a real world scenario of size 450×400 square meters. White areas are contained in Ω_{out} and therefore not walkable.

This means that we need to deal with potential collisions. The parallel update scheme consist of the following steps:

seek: parallel computation of the next desired positions
move: parallel movement of agents and adjustment of their event time if their event time is the smallest among all competing agents

Agents are competing if their bodies overlap with respect to their desired position. These two steps are repeated until all event times are greater than $t + \Delta t$. It is important to notice that Φ_p changes with each repetition. For the first **seek** call Φ_p depends on the agents' positions at time t.

2.3. *Parallel versus Event-driven Update Scheme*

Currently, all OSM parameters are calibrated for the event-driven update scheme. If one wants to use the parallel update for predictive simulations, the parameters must be re-calibrated. The parallel update scheme produces the same result as the event-driven update scheme if **move** only effects one agent, that is, if Δt is sufficient small. In [13] the authors showed that, otherwise, the parallel update scheme produces larger evacuation times. This indicates that agents use sub-optimal paths to their destinations because they lose some of their ability to anticipate other agent's motion. We decided to compare computation times and estimate speed-ups for both schemes.

3. OpenCL Implementations of the Optimal Steps Model

We base our implementation on OpenCL to support a broad range of hardware accelerators. It is integrated in our open source framework Vadere [10] which is written in Java. To call our OpenCL kernels within Java, we use the Lightweight Java Game Library 3.2.3 [14].

The OSM is a model on the operational level. It executes locomotion when each agent's destination is known. Route choice, or selection of the destination, is part of the tactical level, which is, in principle, a decision making process. As such its implementation consists of divergent code paths which does not lend itself to execution on the GPU. Consequently, we focus on the operational level and keep the execution of the tactical level on the CPU.

At the start of the simulation the host (CPU) writes the necessary data (all required agent information, $\Phi_{t,\Gamma}$ and d_Ω) to the device (GPU). The host defines how much time the simulation should be stepped forward by the device. After the device has finished its computation the result is transferred back to the host. This allows us to incorporate the tactical level if necessary.

For the sake of simplicity we assume a constant number of n agents during the simulation which are numbered from 0 to $n-1$ having the same target Γ. To compute the target and obstacle potentials on the GPU, $\Phi_{t,\Gamma}$ and the distance function d_Ω are required. We approximate both by $\hat{\Phi}_{t,\Gamma}$ and $\hat{d}_\Omega$, receptively. They are depicted in Fig. 2. $\hat{\Phi}_{t,\Gamma}$ and $\hat{d}_\Omega$ are solutions of the eikonal equation computed by the Fast Marching Method [15] for a regular grid. In case of the target potential the initial wave front of the Fast Marching Method starts at the target boundary $\partial\Gamma$, i. e., $\hat{\Phi}_{t,\Gamma}(x) = 0$ if $x \in \Gamma$. In case of the distance function, it starts at $\partial\Omega$, i. e., $\hat{d}_\Omega(x) = 0$ if $x \in \Omega_{\text{out}}$. The computation is done on the host. Both grids are transferred into the global memory of the GPU. Values in between grid points are bilinearly interpolated. Furthermore, an approximation of the possible next footstep positions P is computed using a unit circle depicted in Fig. 1 and transferred into cached constant memory. This means that we are using optimizing by "brute force". The possible next positions for agent i at position x_i are given by

$$P_i = \{q \mid p \times \lambda_i + x_i, p \in P\}, \tag{3}$$

where P are the points inside a unit. All required constants such as the domain size and the grid size of $\hat{\Phi}_{t,\Gamma}$ and $\hat{d}_\Omega$ are also transferred to constant device memory. To make use of beneficial coalesce memory, we convert the arrays of structure (AoS), used by the CPU code of Vadere, into a structure of arrays (SoA). Listings 1 and 2 depicts the difference and list all required agent information.

```
class Agent {
  float x;
  float y;
  float eventTime;
  float speed;
  float strideLength;
}
```

Listing 1: Arrays of structure used in object oriented programming.

```
class Agents {
  float[] x;
  float[] y;
  float[] eventTime;
  float[] speed;
  float[] strideLength;
}
```

Listing 2: Structure of arrays used in GPGPU programming.

3.1. The Linked Cell Algorithm

In order to avoid the $\mathcal{O}(n^2)$ complexity of the neighbours search we exploit the locality of agent potentials. Dynamic data structures are difficult to manage on the GPU. The linked cell data structure is a well-known technique to deal with this. We adopt it for our purposes. Let w_Ω, h_Ω be the width and height of a tight bounding rectangle enclosing the whole simulation domain Ω and let c be the cell size of the linked cell data structure, then we divide the space into

$$w_c \times h_c = l, \text{with } w_c = \lceil w_\Omega / c \rceil, h_c = \lceil h_\Omega / c \rceil \tag{4}$$

cells uniquely numbered from 0 to $l-1$. We choose c such that for a given cell, it suffice to consider only agents in its Moore neighborhood to compute the next position of any agent within the cell. Let $v_{\max}, s_{\max}$ be the maximum speed and stride length over all agents. Then a cell size

$$c = \max\{s_{\max}, v_{\max} \times \Delta t\} + r_p + w_p, \tag{5}$$

is sufficient, if we reconstruct the data structure every Δt seconds. Before each update cycle (simulating Δt seconds) we construct the data structure by using an OpenCL implementation of the algorithm described in [16,17] which consist of three steps:

hash: for each agent with position (x,y) its cell id (hash) is computed by $h(x,y) = w_c \times \lfloor y/c \rfloor + \lfloor x/c \rfloor$

sort: agents ids are sorted by a bitonic sort according to their cell id

ordering: agents, i. e., all arrays of the SoA depicted in Listing 2 are reordered according to the sort result

find: cell start indices and cell end indices are detected by unequal consecutive cell ids

The construction is depicted in Fig. 3. The reordering does not only simplify the access to nearby agents but additionally increases the cache hit rate during the following computation steps of the cycle.

3.2. *The Parallel Update Scheme*

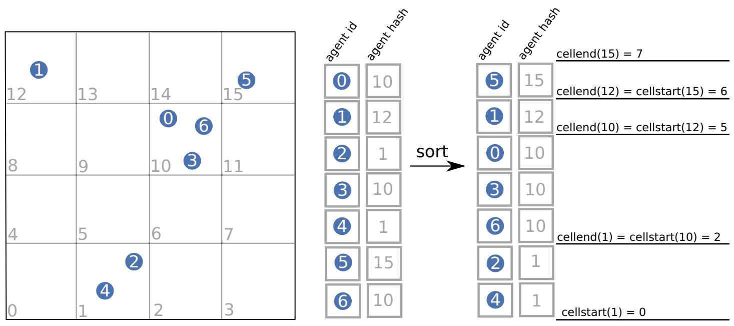
Implementing the parallel update scheme for the GPU is straightforward. One update cycle is realized by invoking multiple OpenCL kernel functions which steps the simulation time from t to $t+\Delta t$. A cycle is completed if there exist no more agent with an event time smaller or equal to $t+\Delta t$. For each agent we have to remember two positions: its actual position and its next possible position. Therefore, we extend the SoA by two additional floating point arrays.

3.2.1. *Seek*

After the linked cell date structure is constructed, we compute the agents' next possible position in parallel. Each agent is assigned to a different work item (thread) executing the **seek** kernel. If the agents' next footstep happens before $t+\Delta t$, that is, if $\tau \leq t+\Delta t$, the next possible best position is computed. The work item reduces all possible positions to the best one by solving Eq. (2). Finally, the resulting position is saved in global memory.

3.2.2. *Move*

For each agent the **move** kernel is executed on a different work item (thread). This kernel tests if there are any collisions with respect to the possible next positions (calculated by


12
13
14
15
8
9
10
11
4
5
6
7
0
1
2
3
agent id
agent hash
sort
cellend(15) = 7
cellend(12) = cellstart(15) = 6
cellend(10) = cellstart(12) = 5
cellend(1) = cellstart(10) = 2
cellstart(1) = 0

agent id	agent hash
0	10
1	12
2	1
3	10
4	1
5	15
6	10

agent id	agent hash
5	15
1	12
0	10
3	10
6	10
2	1
4	1

Figure 3. Construction of the linked cell data structure with a cell width w_c and height h_c equal to 4 and $n = 7$ agents. The spatial domain is covered by the rectangle on the left. Agents numbered from 0 to $n-1$ are depicted in blue. After the sorting based on the agent's hash, the start and end, for example, cell 10 are the two array positions at which the agent hash changes to or from 10, i. e., cellstart(10) = 2 and cellend(10) = 5.

the **seek** kernel) agents within the Moore neighbourhood of the linked cell data structure. If there is none, we update the agents event time

$$\tau \leftarrow \tau + (\lambda_i / v_{i,\text{free}}) \tag{6}$$

and position accordingly. Otherwise, we mark the cycle as conflicted. We repeat the cycle, that is, the **seek** and **move** operation until there is no collision detected and all event times are greater than $t + \Delta t$.

3.3. The Event-driven Update Scheme

The number of agents the event-driven update can update in parallel is greatly reduced compared to the parallel update. An upper bound is given by the number of cells l. And in the worst we can only update a single agent. Therefore, we split the computation of the next agent position into $|P|$ tasks, where $|P|$ is the number of possible next positions of agent i. Let $\mathbf{M}$ contain the agent ids of all agents we can update in parallel. Then we evaluate

$$\Phi_i(x_i + z \times \lambda_i), \text{ for } i \in \mathbf{M}, z \in P \tag{7}$$

in parallel. Beforehand, we have to efficiently compute $\mathbf{M}$ which is realized by the following three kernel functions.

3.3.1. Cellfilter

We implement two filters which are processed consecutively. The first **cellfilter** is invoked for each cell of the linked cell data structure. It iterates over all agents of a specific cell and filters the agent with the smallest event time $\tau \leq t + \Delta t$. Its id is written into an array $\mathbf{M}'$ of size $w_c \times h_c$. If no agent was found, which happens if the cell is empty, -1 is written instead. Compare Fig. 4.

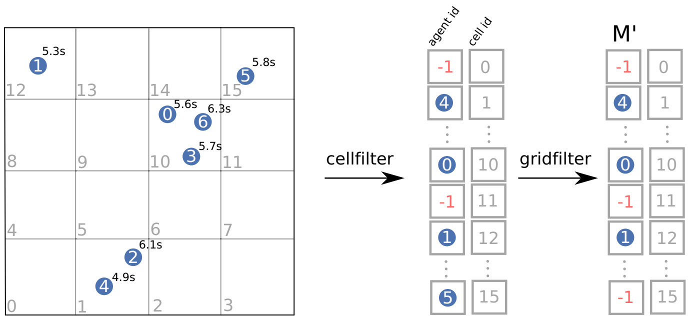


Figure 4. Construction of $\mathbf{M}'$ using the situation depicted in Fig. 3 by invoking **cellfilter** and **gridfilter** consecutively. Blue highlighted numbers represent agent ids and the time represent their event time λ_i. The first array is constructed by **cellfilter**. For each cell the agent with the shortest event time is written into the array. In this example **gridfilter** filters the agent 5 because of the smaller event time of agent 0.

3.3.2. Gridfilter

The second kernel **gridfilter** is also invoked for each cell and filters the left-over agents. It replaces the id by -1 if there is an agent in the Moore neighbourhood with a smaller event time. Note that each work item only has to test 8 agents due to the first filter.

3.3.3. Align

The result of the filtering is a large integer array $\mathbf{M}'$ of size $w_c \times h_c$ containing some agent ids and a lot of negative ones. To compute $|\mathbf{M}|$ we use a modified prefix sum using the algorithm presented in [18]. Instead of summing everything up, we only add 1 if the array value is none negative. Additionally, we compute a second prefix sum array $\mathbf{K}$ ignoring all positive values. Since all agent ids are non negative and all other array entries are set to -1, $-\mathbf{K}[j]$ gives the number of cells with an id smaller than j that are unaffected by any movement update. It follows that $j + \mathbf{K}[j]$ is equal to the number of cells with an id smaller than j which are affected by changes. The **align** kernel is invoked for each cell. Let i be the id of a work item (thread), then all work items generates the aligned array $\mathbf{M}$ of $|\mathbf{M}|$ agent ids by executing the following assignment in parallel:

$$\mathbf{M}[i + \mathbf{K}[i]] \leftarrow \mathbf{M}'[i], \text{ if } \mathbf{M}'[i] \geq 0. \tag{8}$$

After executing the **align** kernel, $\mathbf{M}$ only contains agent ids of the agent which can be updated in parallel. Compare Fig. 5.

3.3.4. Move

To use fast shared memory the **move** kernel is executed by $|\mathbf{M}| \times |P|$ work items grouped into $|\mathbf{M}|$ work groups. The i-th work item (thread) of the j-th work group (thread group) computes $\Phi_{\mathbf{M}[j]}(x_i)$ where x_i is the i-th possible next position. All immediate results are saved into shared memory. Therefore, each work group requires $|P| \times 3 \times 4$ bytes local memory, i. e., 2×4 bytes for each point in P and four bytes to save each evaluation of Φ. After all work items complete their task, the final next position is computed by a parallel reduction using $\lceil |\mathbf{P}|/2 \rceil$ work items which finally solves Eq. (2). The first work item of each work group writes the resulting next position back to global memory. We repeat the cycle, i. e., **cellfilter**, **gridfilter**, **align**, **move** until $\mathbf{M}$ is empty.

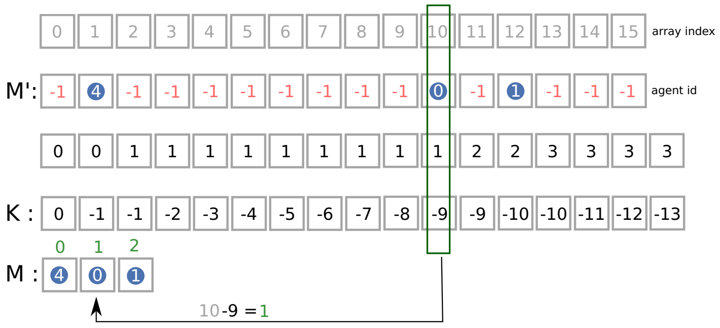


Figure 5. Construction of **M** using the situation depicted in Fig. 3 by invoking **align** after the kernel function **gridfilter** has finished. The third and forth line represent the prefix sum arrays. Blue highlighted numbers represent agent ids.

	parallel update scheme			**event-driven update scheme**		
	OpenCL (GPU)	**OpenCL (CPU**	**Java (CPU)**	**OpenCL (GPU)**	**OpenCL (CPU)**	**Java (CPU)**
10k	3	15	80	20	99	140
100k	13	119	1190	60	546	2100
500k	74	1348	12137	160	2875	30096

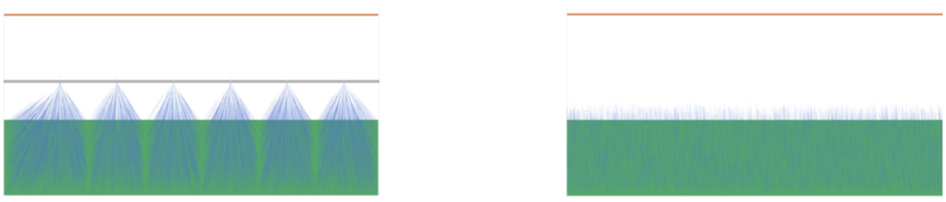
Table 1. Average computation time in milliseconds of $\Delta t = 0.4$ seconds simulation time of the open space scenario for $10 \times 10^3, 100 \times 10^3$ and 500×10^3 agents.

4. Comparison of Computation Times

In order to compare computation times of all implementations, we carry out a series of tests. The parallel event-driven update scheme is expected to perform best for evenly distributed and well-separated agents because in this case their footstep events are likely to be independent from each other. It should perform worst if cells are either empty or highly populated. Therefore, we use two benchmark scenarios. The first one consist of multiple bottlenecks which yield high local densities. Even the multi-bottleneck scenario is simple, it imitates more complex geometries and situations by generating a wide range of densities, i. e., from low densities at the start of the simulation to high densities at the time of congestion. For the second scenario we evenly distribute agents inside a large rectangle at the bottom and place the target at the top. Both scenarios are depicted in Fig. 6. For all tests $|P|$ is approximated by 32 points and Δt is set to 0.4 seconds. Note that our OpenCL implementation uses single precision and the existing Java implementation double precision.

Tests were carried out on the following hardware platform: Intel i5-7400 Quad-Core (3.50 GHz), 8 GB DDR4 SDRAM and a graphics card NVIDIA GeForce GTX 1050 Ti / 4 GB GDDR5 VRAM.

In open space, i. e., for the second scenario, using GPGPU computation over the existing Java implementation speeds up the simulation by multiple order of magnitude, i. e., the simulation runs more than 100 times faster. Running the same OpenCL code on the CPU is $5-18$ times slower compared to the GPU. The GPU scales much better for a growing number of agents. Compare Table 1. Furthermore, during the simulation the computation times do not significantly fluctuate. The multi-bottleneck scenario



a) Bottleneck scneario. **b)** Open space scenario.

Figure 6. Illustration of both benchmark scenarios. All agents are uniformly distributed inside the green rectangle at $t = 0$ seconds. They walk towards their orange target at the top. The blue trajectories reveals the agents' movement through one of the 6 bottlenecks (left) and straight towards their top target (right).

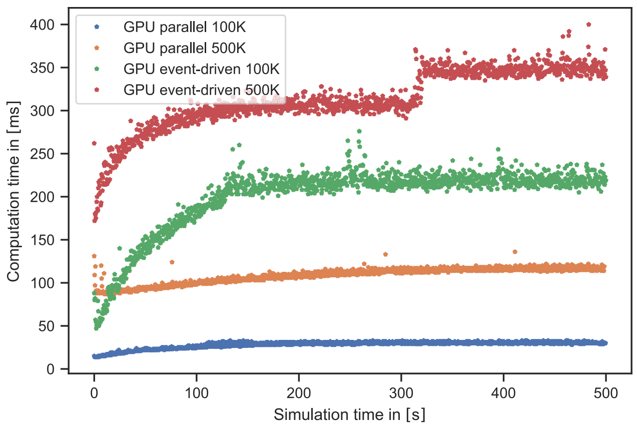


Figure 7. Comparison of computation times over a simulation run of the multi bottleneck scenario for 100 and 500 thousand agents using the parallel and event-driven update scheme. The computation time is required to simulate $\Delta t = 0.4$ seconds.

benchmark reveals that computation times do fluctuate during the simulation run, if the event-driven update scheme is used. As expected, the computation slows down because agents approach the bottlenecks, and thus move closer together. After approximately 100 simulated seconds, the computation time reach a plateau because more and more agents passed the bottleneck. Figure 7 illustrated this phenomenon. The jump at 300 seconds for 500 thousand simulated agents might be the result of some caching effect but further investigations are required.

5. Conclusion

We proposed mechanisms to enable GPU computation for the agent-based Optimal Steps Model which simulates pedestrian dynamics. We presented two implementations: One relied on a parallel update scheme thus modifying the original model. The other parallelized an inherently sequential event-driven update scheme by efficiently identifying independent events and by splitting the event computation into multiple independent tasks. For this we combined a linked cell data structure with massive parallel filtering. We achieved speed-ups of multiple order magnitude for both update schemes compared to the single threaded Java version. Using the same code base but different devices shows that for the chosen hardware setup, the GPU outperforms the CPU by a factor up to 18 for both update schemes. For our specific hardware setup and two non-trivial benchmark scenarios we were able to simulate up to half a million agents faster than real-time. Our techniques can be carried over to any model where the agents' influence remains local and where agents are spatially spread. This is true for many models. Thus we showed that there is great potential in using GPGPU for pedestrian dynamics beyond CA models or differential equation models.

6. Acknowledgement

We thank the research office (FORWIN) of the Munich University of Applied Sciences and the Faculty Graduate Center CeDoSIA of TUM Graduate School at Technical University of Munich for their support. The authors are supported by the German Federal Ministry of Education and Research through the project S2UCRE to study the acceleration of microscopic pedestrian simulations by designing efficient and parallel algorithms (grant no. 13N14463).

References

[1] S. Rybacki, J. Himmelspach, and A. M. Uhrmacher. Experiments with single core, multi-core, and gpu based computation of cellular automata. In *Advances in System Simulation, 2009. SIMUL '09. First International Conference on*, pages 62 –67, Sept 2009.

[2] Q. Miao, Y. Lv, and F. Zhu. A cellular automata based evacuation model on gpu platform. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 764–768, Sep. 2012.

[3] Hubert Mroz and Jaroslaw Was. Discrete vs. continuous approach in crowd dynamics modeling using gpu computing. *Cybernetics and Systems*, 45(1):25–38, 2014.

[4] Jarosław Was, Hubert Mroz, and Pawel Topa. Gpgpu computing for microscopic simulations of crowd dynamics. *COMPUTING AND INFORMATICS*, 2015.

[5] Adrian Kulusek, Pawel Topa, and Jarosław Was. Towards effective gpu implementation of social distances model for mass evacuation. Working paper, 2016.

[6] Adrian Kłusek, Paweł Topa, and Jarosław Was. An implementation of the social distances model using multi gpu-systems. Working paper, 2016.

[7] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. A survey on agent-based simulation using hardware accelerators. *ACM Comput. Surv.*, 51(6):131:1–131:35, January 2019.

[8] Michael J. Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86(4):046108, 2012.

[9] Isabella von Sivers and Gerta Köster. Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74:104–117, 2015.

[10] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, and Gerta Köster. Vadere: An open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4, 2019.

[11] Michael J. Seitz, Felix Dietrich, and Gerta Köster. The effect of stepping on pedestrian trajectories. *Physica A: Statistical Mechanics and its Applications*, 421:594–604, 2015.

[12] Isabella von Sivers and Gerta Köster. How stride adaptation in pedestrian models improves navigation. *arXiv*, 1401.7838(v1), 2014.

[13] Michael J. Seitz and Gerta Köster. How update schemes influence crowd simulations. *Journal of Statistical Mechanics: Theory and Experiment*, 2014(7):P07002, 2014.

[14] Lightweight java game library 3. https://www.lwjgl.org/.

[15] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[16] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 51–58, Oct 2009.

[17] Simon Green. Particle simulation using cuda, May 2010.

[18] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.