Postprint of: Skrzypczak J., Czarnul P., Efficient parallel implementation of crowd simulation using a hybrid CPU+GPU high performance computing system, SIMULATION MODELLING PRACTICE AND THEORY Vol. 123 (2023), 102691, DOI: 10.1016/j.simpat.2022.102691

Efficient parallel implementation of crowd simulation using a hybrid CPU+GPU high performance computing system

Jakub Skrzypczak[a], Paweł Czarnul[a,]

[a]*Department of Computer Architecture*
Faculty of Electronics, Telecommunications and Informatics
Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland

Abstract

In the paper we present a modern efficient parallel OpenMP+CUDA implementation of crowd simulation for hybrid CPU+GPU systems and demonstrate its higher performance over CPU-only and GPU-only implementations for several problem sizes including 10 000, 50 000, 100 000, 500 000 and 1 000 000 agents. We show how performance varies for various tile sizes and what CPU-GPU load balancing settings shall be preferred for various domain sizes among CPUs and GPUs of a high performance system with 2 Intel Xeon Silver multicore CPUs and 8 NVIDIA Quadro RTX 5000 GPUs. We then present how execution time depends on the number of agents as well as the number of CUDA streams used for parallel execution of several CUDA kernels. We discuss the design and implementation of an algorithm with CPU computational threads, GPU management threads, assignment of particular tasks to threads as well as usage of pinned memory and CUDA shared memory for maximizing performance.

Keywords: crowd simulation, parallelization, hybrid CPU+GPU system, high performance computing

Email addresses: jskrzypczak97@gmail.com (Jakub Skrzypczak), pczarnul@eti.pg.gda.pl (Paweł Czarnul)

1. Introduction

Crowd simulation has become an important topic in today's society as it allows to model many scenarios that are of interest especially in modern cities – in the context of evacuations because of natural disasters, facility failures, terrorist attacks as well as optimizing traffic, designing new buildings, areas, passes etc. As the density of objects in such scenarios is still increasing, use of high performance computing systems [1] has become a necessity to simulate a large number of possibly interacting agents [2, 3, 4], using also new hardware solutions such as NVRAM for efficiency and fail-safety of a solution [5].

Currently most of computer systems are available in multicore CPU(s)+ GPU(s) setups which on one hand made it possible to exploit computational power of such a hybrid system, on the other hand this task is difficult as it requires non-trivial load balancing, minimizing communication and overheads and mastering proper APIs. CPU+GPU systems have been known to deliver high performance for various applications such as: matrix factorization [6], optical remote sensing image processing [7], parallelization of large vector similarity computations [8] etc. The contribution of this work is a modern efficient CPU+GPU OpenMP+CUDA implementation of the demanding crowd simulation problem that demonstrates performance better than CPU and GPU only solutions using techniques such as load balancing with tiling, multiple CUDA streams, pinned and shared GPU memory.

2. Related work

In the literature there have been several models [9] proposed for crowd simulation that can be grouped into major categories such as: microscopic (rule-based, force-based, velocity-based, agent-based and vision-based) [10], macroscopic (continuum models, aggregate dynamics, potential field-based models) as well as mesoscopic models (dynamic group behavior, interactive group formation, social psychological crowds). In terms of future challenges and directions such as generally increasing realism of simulations [11], apart from approaches and improvements such as data collection and integration, data-driven crowd modelling [12, 10], cognitive-based crowds, learning to generate character animation [10], using machine learning approaches, computational efficiency through parallel computing is of key importance in order to simulate large-scale realistic crowds.

There are many examples of solutions that use a parallel environment for crowd simulation. The most common motivation for their development is the desire to deliver an implementation that could handle a very large number of agents while providing reasonable execution time of the application. Ways of solving the problem of crowd simulation parallelization are different. This is due to differences in the assumptions made during implementation or slightly different goals that their authors have set for themselves. Nevertheless, all existing solutions can be assigned to one of three categories which are CPU-only solutions, GPU-only solutions and CPU+GPU solutions.

2.1. CPU solutions

It appears that most available parallel solutions focus only on using CPU multithreading. In such implementations, the most commonly used interfaces are OpenMP and MPI, the latter developed for inter-node communication in clusters. Results of the tests performed on each implementation show that parallelization on the CPU does indeed improve the performance of crowd simulations, which is particularly noticeable for large numbers of agents. Typically, in these types of solutions, simulations are run for a couple or tens of thousands of agents. An exception is work [2], where the heaviest test cases involved 250 000 individuals at the same time. The scalability of the solutions varies. In works [13, 14] it was shown that increasing the number of resources does not always contribute to an increase in program performance. On the other hand, the results of the implementation presented in paper [15] suggest a linear relationship between computing resources and number of simulation updates per second (which was the performance evaluation metric in this case). In this solution type, the crowd behavior model often plays the main role. The behavior of individuals is often simulated using complex models that emphasize realism. In such cases, the matter of parallel processing is treated as a secondary issue; it is introduced to ensure acceptable performance of the computations.

2.2. GPU solutions

Solutions based on GPU computations present a slightly different approach to the problem. Typically, their goal is to run simulations for many hundreds of thousands or even hundreds of millions of agents in real-time. Such an assumption implies the need for simplifying the crowd behavior models used in the simulation quite significantly. Implementations [16, 3] are based on the assumption that agents can only navigate between predefined

environment cells, which is rather unrealistic. Nevertheless, the presented solutions are able to achieve high computational efficiency, which was their main goal. Authors of such solutions often mention scalability as their main advantage, which is noticeably higher than for CPU solutions [3]. Typically, a linear increase in simulation execution time is observed with an increasing number of agents involved in a given scenario. In many cases, the use of more than just one GPU device is considered when designing solutions of this kind. Such an approach was proposed in work [16]. By using multiple GPU clusters, its authors managed to run simulations considering 529 million individuals. Interestingly, in this solution using a relatively large number of nodes for computations resulted in a significant increase in inter-node communication time. For this reason, the highest performance was achieved when not all available GPUs were used.

2.3. Hybrid solutions

CPU+GPU hybrid solutions are typically structured in a similar way to GPU solutions; they use less complex models to support as many agents as possible. An example of this approach is work [4], which is actually an extension of work [16] presented in Section 2.2. This implementation is based on MPI, CUDA and the OmpSs interface which is an extension of OpenMP. Thanks to the combined power of the CPU and GPU, authors of the solution managed to handle up to 764 million agents at once. Authors of paper [17] decided to use the GPU in a slightly different way than in the previously mentioned implementations. Instead of CUDA or OpenCL, GLSL (OpenGL Shading Language) [18] and textures were used to perform appropriate calculations on the graphics card. In a hybrid approach, it is crucial to properly design how the processor and graphics cards work together. It is essential to be aware of the advantages and disadvantages of each side. For example, the CPU is good at complex calculations containing many conditional instructions etc. On the other hand, its limitation is a relatively small number of cores. In the case of the GPU, the opposite is true. Graphics cards have many cores, which allow for massive parallelization, but the best performance is obtained for a code without or with limited divergence. In addition, an important aspect that has a significant impact on the performance of CPU+GPU solutions is the data transfer time from the processor to the graphics card and vice versa. Often these operations are the bottleneck of the algorithm so minimization of communication is essential.

3. Proposed approach

3.1. Crowd behavior model

In this solution, Helbing's model [19], classified as a popular social force model [20], is used to simulate crowd behavior. It assumes that the movement of agents is described by three components: self-driving force with current velocity and agent mass considered, external force exerted by all neighboring agents, and external force exerted by all neighboring obstacles. The sum of these values gives the resulting force on the basis of which it is possible to determine new positions and velocities of the agents.

The decision to choose this particular model was motivated by the desire to achieve a certain compromise between realism and simplicity, in contrast to more complex models e.g. Xu's work [21] – in which an agent's behavior is influenced by their gender, age or heart rate level or Mohd Nasir's work [22], which involves the implementation of relatively advanced behavioral mechanisms such as group formation. Helbing's model is a proven solution that was developed by experts in the field. It allows for reproducing many behaviors and phenomena that can be observed in real life scenarios, thanks to which the simulations based on this model can be considered realistic. On the other hand, the idea behind Helbing's model is relatively simple. The model does not take very small details concerning agents' representation or behavior into account, which for scenarios involving a large number of agents would have an insignificant impact on the credibility of the simulation. As an example of a more complex model, work [21] can be mentioned, in which an agent's behavior is influenced by their gender, age or heart rate level. A different example can be work [22], which involves the implementation of relatively advanced behavioral mechanisms such as group formation. Of course, such details indeed increase the level of realism of the solution, however, they usually have a negative impact on performance as well, which could be particularly noticeable for large-scale simulations. Moreover, it might not be irrelevant that the approach described in this work relies greatly on graphics card computations, which require a slightly different programming approach to fully exploit GPU potential. For example, it is recommended to avoid conditional statements (*if else* statements) in the logic of GPU kernels as they may lead to the so-called branch divergence which has a major impact on performance [23]. In the implementation of more complex models, avoiding these types of instructions could be very difficult or impossible, which could consequently reduce the usefulness of GPUs usage significantly. In terms

of performance, the most interesting are relatively simple models that are based on using a discrete domain. For instance, Pérez's solution [4], which is capable of handling hundreds of millions of agents simultaneously, is one of such approaches. However, these types of models lack realism and for this reason they have not been considered.

3.2. Navigation system

In Helbing's model, each agent moves toward a certain destination point that is assigned to it. In the simplest simulation scenarios it would be sufficient to assign one fixed target point to every agent at the beginning of the simulation but for more complex cases this approach would be limiting. For this reason, we decided to develop a navigation system that would allow new target positions to be dynamically assigned to agents during simulation. Such a solution does not interfere with the way the model works; it is only an extension thanks to which the agents can navigate in more complex environments.

The issue of agent navigation in similar solutions has been addressed in different ways. For example, in work [24], the A* algorithm [25] was implemented for navigation, which allows finding the shortest path from point A to point B. Despite the high efficiency and reliability of this algorithm, it requires performing an environment space search which is costly in terms of performance. A different approach was presented in [26]. In this solution, the decision in which direction an agent should move is determined by its position in the environment. This method of navigation has some limitations. For example, it is not possible to create a scenario in which agents residing in a certain space move in opposite directions. Nevertheless, a great advantage of this approach is its simplicity and minimal impact on simulation performance. For this reason the decision was made to develop a solution based on this approach.

The developed agents' navigation system mainly consists in defining so-called navigation zones in the simulation environment. A navigation zone is a rectangular space of an arbitrary size that navigates all agents within the zone to the same navigation point. This means that when an agent crosses the boundary of a navigation zone, its destination point is changed to the navigation point assigned to that zone. Each navigation zone is assigned exactly one navigation point. One navigation point can be assigned to more than one navigation zone. Figure 1 shows the principle of the navigation system. Each navigation zone is marked with a different color. A navigation

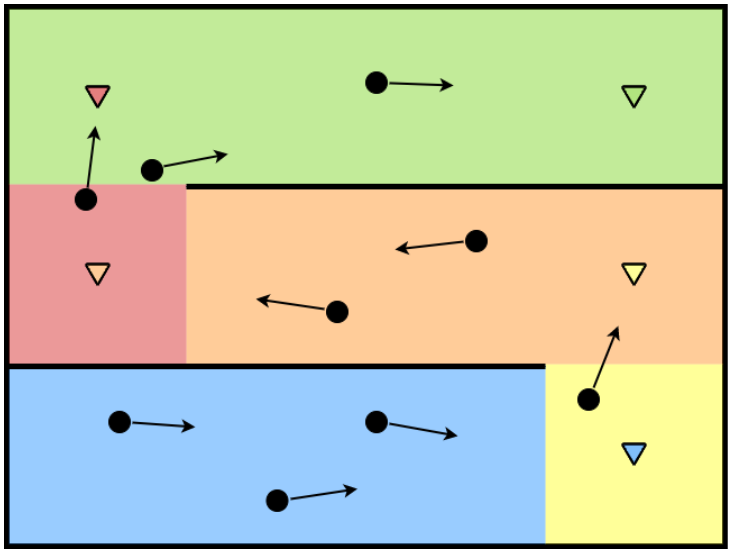

Fig. 1: The principle of the agents navigation system.

point that is assigned to a zone is marked with a triangle of the same color. Agents are represented by black dots.

3.3. Parallelization approach

For parallelization of simulation computations we adopted simulation environment partitioning. This approach is widely used in similar solutions, as evidenced by the work of Pérez [4] and Quinn [15]. In our solution, partitioning of the environment is performed by defining a so-called computational grid (also referred to as a grid). A single element of the grid is called a computational tile (also referred to as a tile). Each tile is a subspace of the simulation environment and therefore has a specific position and size. Its purpose is to identify a group of agents whose state can be updated in an independent way by one of the threads running within the application. In order to provide such computational independence, it is necessary to obtain data on agents located in neighboring tiles. In Quinn's and Pérez's solution, this problem was solved by performing communication and mutual information exchange between independent regions. In our solution exchange of information takes place indirectly by the means of so-called margins. Margins are additional spaces defined around a given tile which overlaps all neighboring tiles. Thanks to them each tile is aware of the presence of the agents in its immediate proximity, who may have a significant impact on the state of the

agents belonging to the tile (Figure 2). It is important to note that it is not the tiles but a management thread that is responsible for the content of the margins. This means that there is never any direct communication between tiles. Updating the content of the tiles and their margins with the latest data is one of the operations performed (in each step of the simulation) before the actual processing, thanks to which the tiles are fully independent from each other at the processing stage.

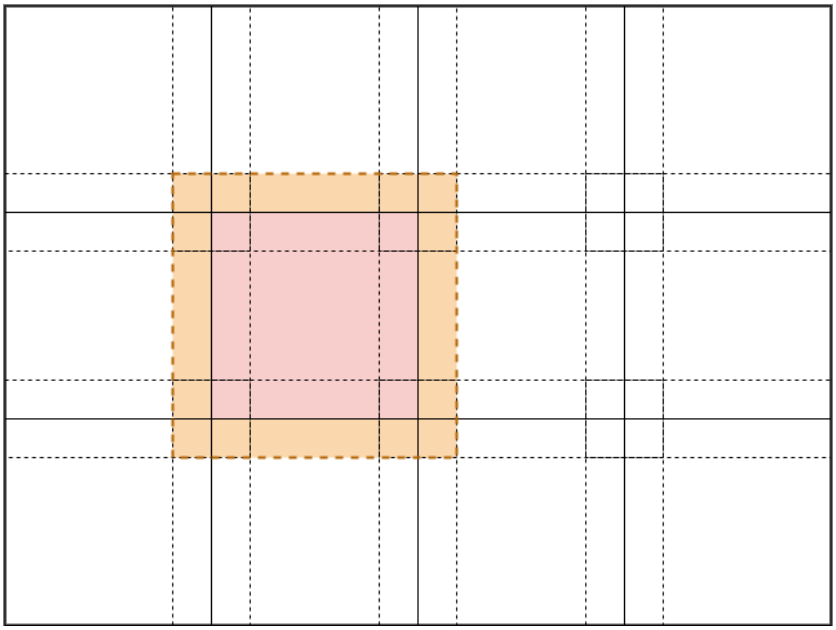

Fig. 2: Example of partitioning the simulation environment into computational tiles. One of the tiles is highlighted. The red color indicates the area of the tile, and the orange color indicates the area of its margins.

Each computational tile has data collections associated with it: identifiers of so-called primary agents, identifiers of so-called secondary agents, and a collection of positions of all obstacles located within the tile and its margins. Primary agents are the individuals whose state must be updated by a thread to which the tile was assigned. Usually this term is used for agents that are located inside the tile. Secondary agents are the individuals whose presence is important from the point of view of performed calculations, but their state is not updated during tile processing. These agents are treated as obstacles that may be important in determining new states for primary agents. Usually these agents would be located in the margins of the given tile. The exception

for this are the agents that are inside of the tile, but were injured in previous steps of the simulation. This is because the state of once injured agents is not being updated anymore. The collection of obstacle positions plays a similar role as the collection of secondary agents identifiers, however, contrary to it, the content of the collection of obstacle positions does not change throughout the simulation.

An important aspect of computational tiles is that their size is not predefined. Their width, height and margin size are input parameters. The size of the tile determines how work is distributed among the available threads/devices. Selecting a smaller tile size will result in the creation of a large number of tiles whose content is relatively light computationally. On the other hand, large tile sizes will result in creating a small number of tiles which would usually be relatively heavy computationally. This means that the size of the computational tile can have a significant impact on overall performance. When choosing the tile size, it is worth considering the number of agents involved in the simulation, the size of the simulation environment and the available resources.

To update the state of each primary agent it is necessary to have information about every other primary agent, about every secondary agent and about every obstacle located inside the tile or on its margins. Access to such a set of information guarantees that the thread processing a given tile will be able to correctly update the state of all primary agents. However, it is important to note that not all agents and obstacles inside the tile (or on its margins) have a significant impact on the state of a given agent. It can be observed that the larger the distance between a given agent and its neighbor/obstacle, the smaller the force that affects it. What is more, at a sufficiently large distance, the effect of such a force is virtually negligible and for this reason its value could be fully omitted. The parameter describing the limiting value of the distance beyond which the force is considered negligible was called the radius of influence. This means that if during the agent's state update process the second agent or obstacle is located at a distance greater than radius of influence then the given force will not be calculated at all (Figure 3).

3.4. Algorithm overview

The main loop of the algorithm is executed in parallel by multiple threads using OpenMP, similarly to design of a parallel CPU+GPU framework in [27], where threads with ids in the range [0, *number of available GPU devices*)

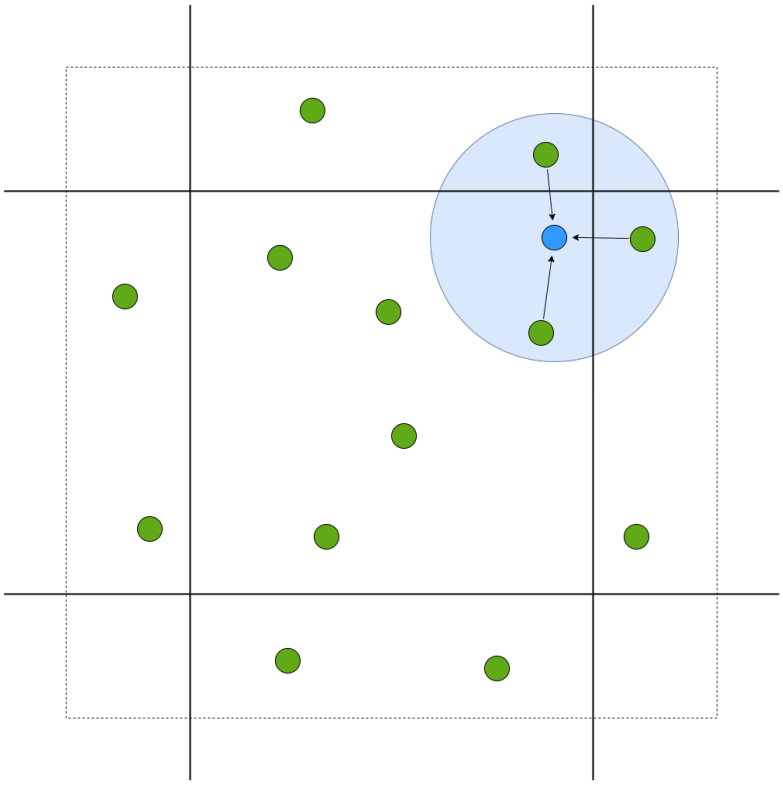

Fig. 3: The effect of using the radius of influence on the number of agents considered when updating the state of a given agent. Solid lines represent the boundaries of the computational tile, dashed lines define its margins.

manage GPUs, and the other threads participate in operations taking place on the CPU side. A GPU managing thread handles only one and the same device throughout the simulation. Inside the loop, there are three stages of computations execution. All operations in a given stage must be completed before moving to the next stage. For this reason, at the end of each stage all operating threads are synchronized. Each stage of the main loop will now be briefly described.

3.4.1. Stage I

At this point three independent operations are performed in parallel, as shown in Figure 4: updating the data on the GPUs with determining the self-driving force of each agent, reassigning agents to the computational tiles, and saving the current position of the agents computed in the previous

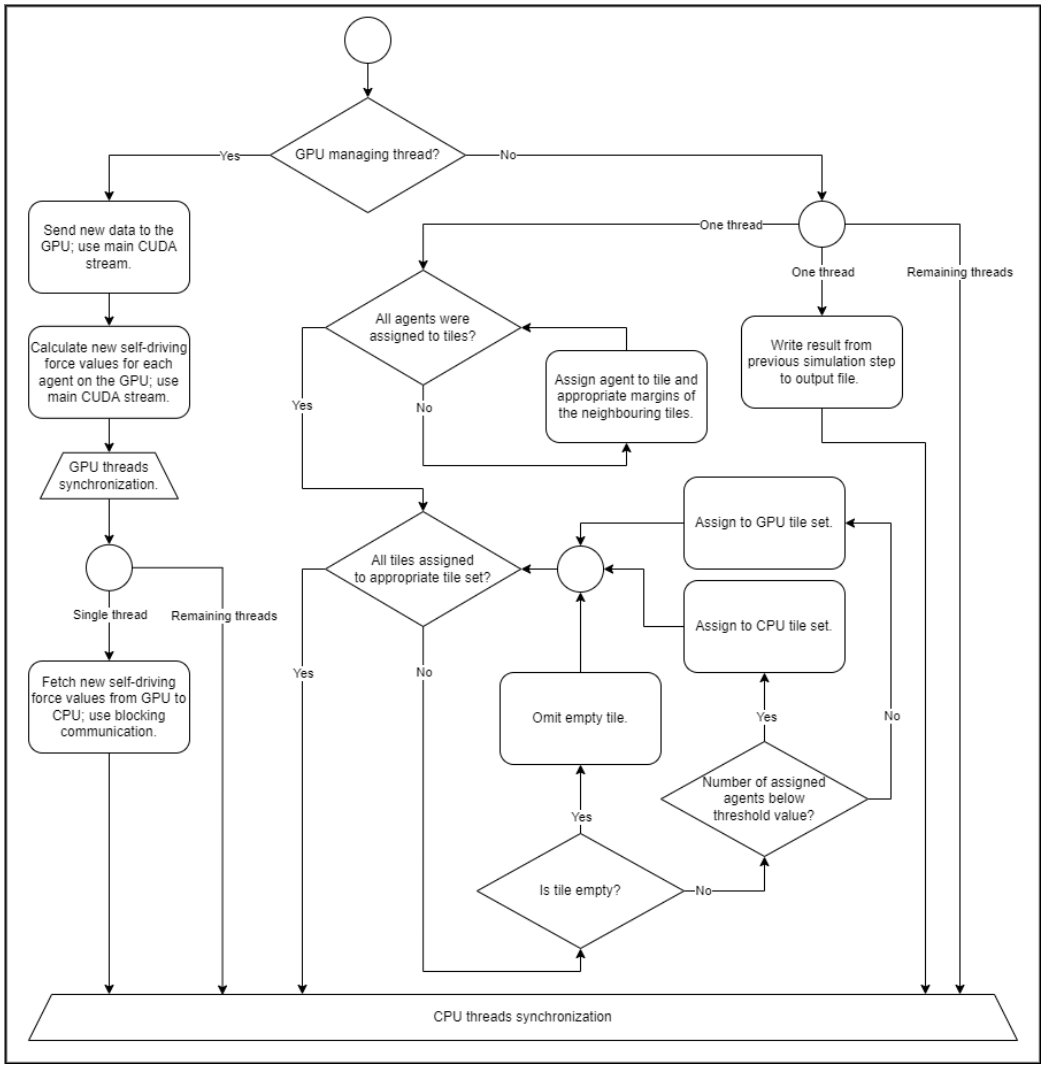

Fig. 4: Diagram illustrating Stage I of the main program loop.

simulation step in the output file. If the number of created threads is large enough, all these operations can be performed in parallel. Updating data on GPU devices and calculating new self-driving force values is performed only by the threads designated for device management. First, the most recent states of the agents that were determined in the previous simulation step are sent to each device. This is necessary because in order to update the agent states the computing unit (GPU device or a CPU) must have access to

all agent states from the previous simulation step. Data is transferred to a location in the memory of the devices that holds the agent states designated for reading. The next step is to compute new self-driving force values for each agent based on their states from the previous simulation step that have just been transferred to the device. Afterwards, self-driving force values from the device are transferred to the host to make the results available to the threads operating on the CPU side. The last operation performed by all GPU management threads is device synchronization so that all scheduled tasks are completed.

The assignment of agents to computational tiles is performed by a single CPU thread. The main reason for this is that the agents assignment operation involves performing a large number of writes to shared resources, which, when using multiple threads, would require very frequent synchronization between them, significantly affecting performance. Firstly, based on the agent's position, the thread finds a tile to which the agent should be assigned, and after that the agent's belonging to the margins of neighboring tiles is determined. Based on that information, the agent's id is added to the collection of secondary agent identifiers of the corresponding neighboring tiles. The final step is to split the computational tiles into two sets for load balancing of computations: the sets of tiles to be processed on the CPU(s) and on the GPU(s) respectively. The division is done based on the number of agents assigned to a tile and a certain threshold, which is one of the simulation parameters. If the number of primary agents assigned to a given tile is greater than or equal to the threshold value, the tile will be added to the set of tiles designated for GPU devices, otherwise to the set of tiles designated for CPU threads. If there are no primary agents on a given tile, that tile is not assigned to either set and its content is cleared. Splitting tiles into these two sets is done in order to balance computations between the host and the devices.

3.4.2. Stage II

The second stage, depicted in Figure 5, focuses primarily on processing all of the computational tiles. The way tiles are processed is different for device managing threads and for regular CPU threads. For GPU management threads, the second stage begins with entering the loop which will be executed until all device tiles have been processed. A thread fetches a tile in a critical section and starts processing it. In the case of GPU threads, processing consists primarily of appropriate transfer of data regarding the agents

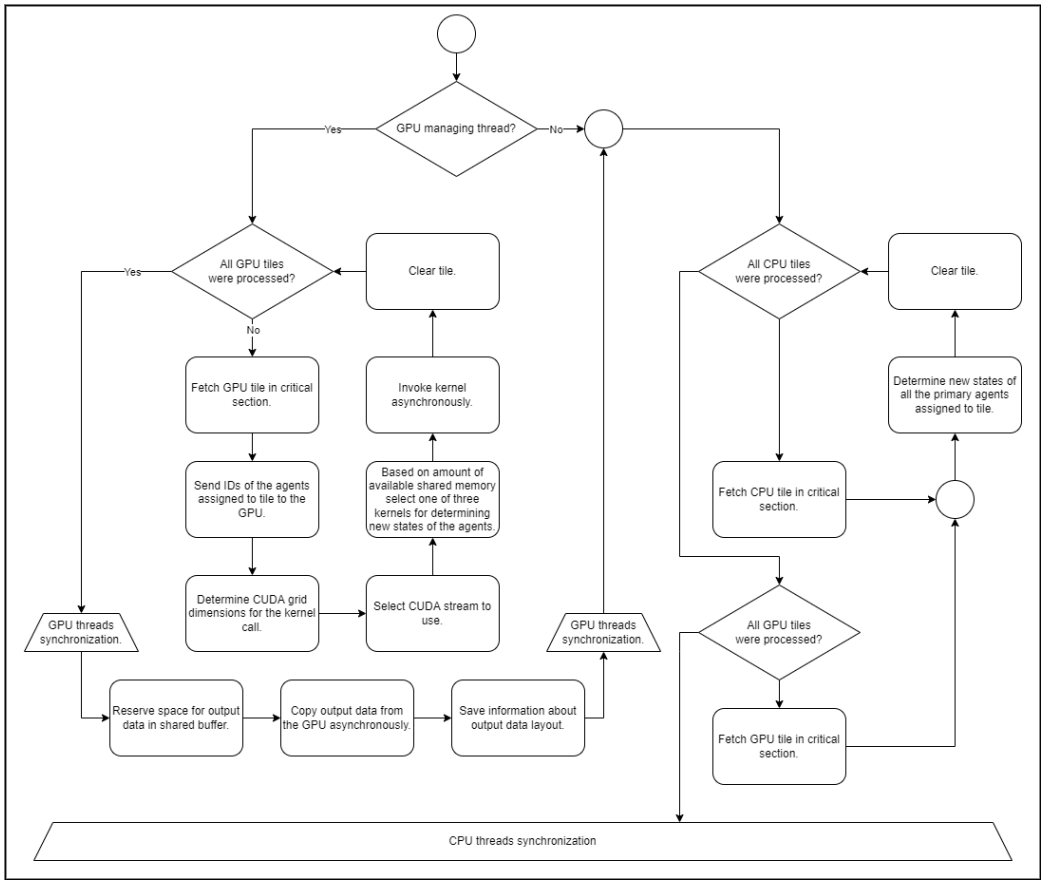

Fig. 5: Diagram illustrating Stage II of the main program loop.

assigned to the tile and the invocation of the kernel responsible for performing simulation calculations. The only information that the GPU needs to properly update the state of the agents from a given tile is the collections of primary and secondary agents' identifiers. Both of these are copied to a specially allocated buffer located in the global memory of the device. After the data has been transferred, the kernel call parameters/configuration is determined. This includes selection of CUDA grid size for the kernel call (number of blocks and threads per block), choosing an appropriate kernel variant depending on the amount of shared memory (three are available) available on a GPU and determining which CUDA stream should be used for execution. The use of CUDA streams is a very important part of the solution, as they allow for much better utilization of the GPU's potential. In this approach it

is a relatively undemanding task for the GPU to process a single computational tile, which means that only a small fraction of the device's resources will be occupied during such an operation. To overcome this problem we use multiple CUDA streams so that several computational tiles can be processed in parallel which provides a significant performance speed-up. This matter will also be discussed in the following sections. The new agent states computed within the kernel are stored in the dedicated buffer in the device memory. The order of the states is the same as the order of primary agents ids that were earlier provided. For this reason, the GPU managing thread, saves these values on the host so that the results can be correctly assigned to the appropriate agents. When all tiles are processed, threads leave the loop and start the process of fetching results from devices to the host. Once this process is complete, the device management threads join the CPU threads to help them process tiles if there are still host tiles available.

The process of distributing tiles among CPU threads is similar to GPU threads to some extent. It means that after entering the loop the CPU threads will fetch the tiles designated for them first in a critical section. If all tiles in the set have already been processed, the CPU threads will start fetching GPU tiles to help with the work. Processing of the tile by a CPU thread is different from that of the GPU thread, because the thread must perform all the necessary simulation computations, rather than simply delegating this task to the device. Processing begins with reading the states of all agents assigned to the tile (both primary and secondary agents) in a critical section from a shared resource into private buffers to avoid conflicts with other threads during the computations. Such prepared data is then used to determine the new states of all agents. Results of the calculations are written to the corresponding buffers.

3.4.3. Stage III

The third stage, visualized as an activity diagram in Figure 6, focuses on merging the results obtained on the GPU side with the results of the CPU computations. Since the results from all devices are in the same buffer it is possible to parallelize these operations among all created threads using the *pragma omp for* directive. Apart from merging results, agents whose state was updated on the GPU are assigned new navigation points. This was not done earlier because the navigation system is not available within the kernels. We made such a decision due to the fact that finding navigation points involves using loops and several if statements that can significantly inhibit

GPU performance. For this reason, determining new navigation points for agents can only be done by the CPU thread. After the third stage has been completed, the algorithm goes back to the first stage and all the aforementioned operations are performed again using new states of the agents.

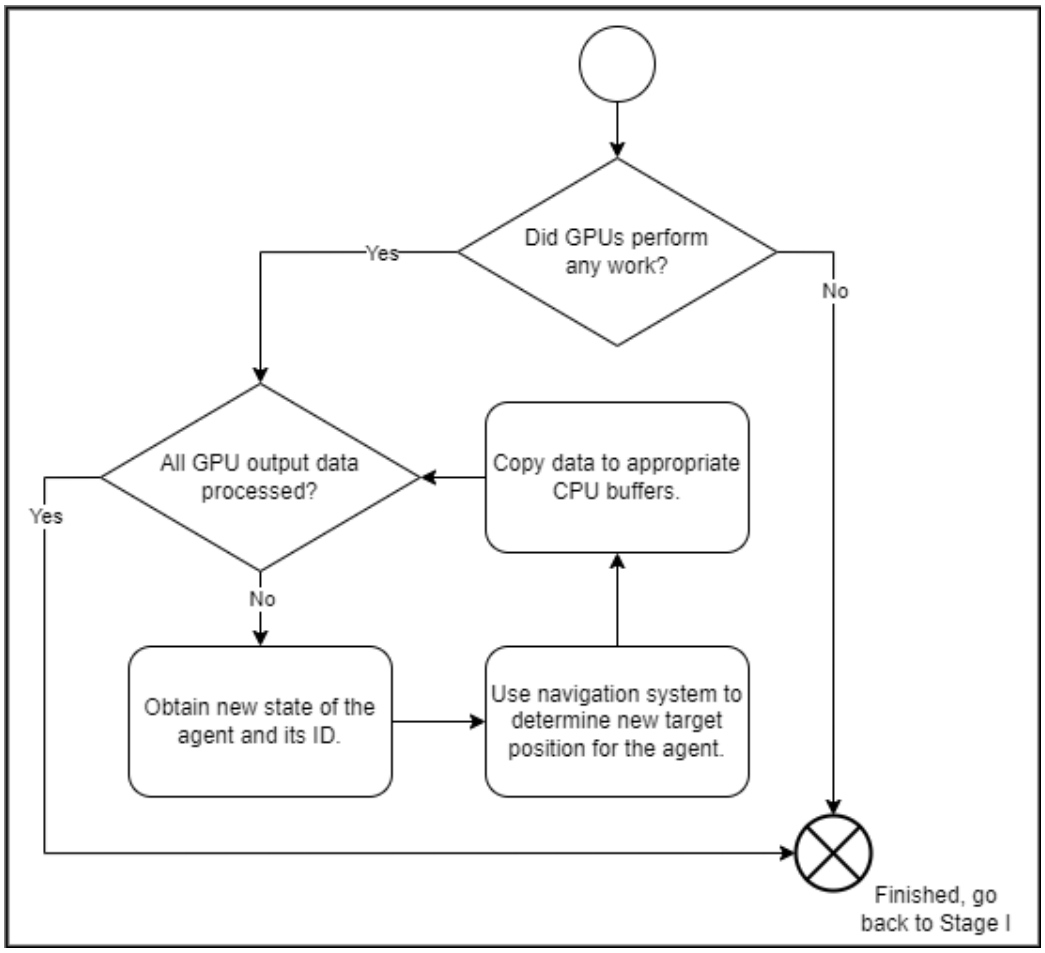

Fig. 6: Diagram illustrating Stage III of the main program loop.

3.5. Optimizations

Apart from optimizations already mentioned when discussing the algorithm, we have also implemented several improvements related to various aspects such as the organization of computations, reducing the idle time of threads or the use of additional functionality of the APIs. The following

subsections will briefly discuss the most important optimizations that were implemented in the solution.

3.5.1. Balancing the workload between CPU and GPU

For efficient computation management a workload balancing mechanism has been developed which, after updating the assignment of agents to the computational tiles, divides all of them into two sets: a set of tiles designated by default for processing on the CPU side and a set of tiles designated by default for processing on the GPU side. The decision on which set a tile will be placed is based on the number of agents assigned to it and a threshold value which is one of the simulation parameters ('GPU tile threshold'). If the number of assigned agents is greater than or equal to the threshold value, then the tile is assigned to GPU-side processing, otherwise to CPU-side processing. This is primarily because the GPU is more efficient for more computationally demanding operations. If there are only a few agents on a tile, the overhead caused by having to transfer data between the CPU and GPU makes processing such a tile on the device side slower and inefficient. It is also important that the process of splitting tiles into sets rejects empty tiles. If multiple threads are running within the application, the waiting time to obtain a tile is relatively long. In this case, it is undesirable for a thread to reject a tile after a long wait in the queue due to lack of computations to perform on it.

3.5.2. Multiple CUDA streams usage

In general CUDA streams help to achieve better GPU utilization in two ways. They allow overlapping in 2+ streams data transfer from host to device (and vice versa) and simultaneous execution of multiple kernels. The second property proved to be particularly important in the implementation of this solution. In general, whether the use of streams results in parallel execution of multiple kernels depends on the amount of available device resources. The more available resources, the more kernels can be executed in parallel. Since GPU devices are designed for massive computations, a single computational tile is usually a relatively small resource workload. It means that by using streams, a sufficiently powerful GPU is able to process multiple tiles simultaneously. In our solution, each device has its own pool of CUDA streams, which are used to invoke kernels during algorithm execution. When a computational tile is selected for processing, the thread is assigned the CUDA

stream that has not been used for the longest period of time. This ensures that all streams in the pool are used evenly.

3.5.3. Pinned memory usage

The implementation also uses a special type of memory called pinned or page-locked memory [28]. Using the standard page-locked host memory would require allocation of a temporary page-locked buffer and copying data to the GPU through that buffer. When using the pinned memory, which is a prerequisite allowing asynchronous data copying, data to and from the GPU can be copied much faster and directly. In the solution, pinned memory was used to transfer current agent states to the device and to fetch new self-driving force values from the device. Furthermore, zero-copy memory [28] was used for primary and secondary agent ids which allows data to be made available to the device without the need for an explicit transfer operation.

3.5.4. Shared memory usage

Shared memory is a special type of device memory that allows very fast reads and writes from/to shared data within a thread block [28]. Reading from shared memory is multiple times faster than reading from the global memory. When properly used, shared memory can bring great performance benefits. Typically it is used as a cache, even allowing prefetching data from global memory, when used in conjunction with registers. We used it in our kernel implementations, even providing several kernel variants depending on the amount of memory available on the device. In Stage I of the main loop, if a given device has enough shared memory a kernel using shared memory is used for computing new self-driving force values for each agent, otherwise a version without shared memory is used. In Stage II of the main loop, we have implemented several kernel versions for updating agents' states. These differ in the intensity of shared memory usage. In the first variant, shared memory is used to store the states of primary and secondary agents, in the second variant it is used to store only the states of primary agents, and the third one does not use shared memory at all. Depending on the specifications of a GPU, a variant is chosen in this very order, starting from the first one.

3.5.5. OpenMP loop usage

The loop parallelization mechanism provided by the OpenMP interface was used in the process of merging the results obtained on the CPU side with the results obtained from the GPU. It allows the use of all threads created

within the *#pragma omp parallel* directive, instead of just device management threads. This makes the results merging operation more efficient which is especially noticeable when the states of a large number of agents have been updated by GPU devices.

3.5.6. Data duplication

The data duplication that we used in the implementation was primarily intended to minimize the number of time-consuming thread synchronizations resulting from access to shared data. This allowed multiple threads to simultaneously perform tasks based on the same input data (e.g., operations performed in Stage I of the main program loop). It is worth mentioning that data duplication carries some performance overhead, but it is generally relatively small compared to the overhead resulting from the need to synchronize threads when certain resources are shared between them.

4. Experiments

The experiments mainly focused on verifying the performance of the application for simulation scenarios of different sizes and comparing it with the performance of non-hybrid implementations that have been specially prepared for the tests.

4.1. Testbed environment

Tests were conducted on a high performance workstation equipped with two Intel Xeon Silver 4210 processors running at 2.20GHz and 376GB of RAM with a total of 40 logical cores. In addition, the machine has eight NVIDIA Quadro RTX 5000 graphics cards with 16GB of GDDR6 memory each. The machine runs on Debian GNU/Linux 10 OS (buster). The source code was compiled using GCC 8.3.0 and NVCC 10.1 compilers.

4.2. Tests

4.2.1. Hybrid implementation performance analysis

Testing of the hybrid CPU+GPU solution was conducted for five simulation scenarios that differ in the number of participating agents. In Scenario I 10 thousand agents are involved, in Scenario II 50 thousand agents, in Scenario III 100 thousand agents, in Scenario IV 500 thousand agents, and in Scenario V 1 million agents. Testing was largely based on finding configuration of simulation parameters for each Scenario resulting in efficient execution. In particular, we focused on the following:

- Size of calculation tiles. The tiles were assumed to be square in shape with a margin of 2m. For each Scenario, tiles of sizes 8x8x2, 10x10x2, 15x15x2, 20x20x2, 25x25x2, 30x30x2, and 40x40x2 were considered. These values were selected based on the results of early testing, which allowed us to determine tile sizes for which it is possible to obtain the interesting results for all considered Scenarios.

- The number of CPU threads used. Values of 10, 20, 40, 80 were tested for each Scenario.

- The number of GPU devices used. For each Scenario, values of 1, 2, 4, 8 were tested.

- GPU tile threshold – the value based on which the decision is made whether to process a tile on CPU or GPU (described in detail in Section 3.5.1). Different values were chosen for each Scenario due to the different density of agents on the tile in each Scenario. The only parameter that takes different values depending on the Scenario. The motivation behind this approach was to narrow down the number and the range of values tested in order to obtain more interesting results without examining impractical configurations. It is important to recognize that the threshold value should be tailored to the number of agents that can fit within a tile of a given size. For example, using a relatively large threshold value for the tiles of small size would result in a vast majority (or all) of the tiles being assigned to CPU threads. The inverse relationship is also true. Selecting appropriate values for this parameter was done during early testing similarly to the parameters related to tile size. This process consisted of gradually narrowing the threshold values for each Scenario in such a way that the initial results obtained were the best possible. Based on these results, a range consisting of four values that showed the greatest potential in terms of performance was selected.

Another very important parameter from the performance point of view is the number of CUDA streams per device. In case of this parameter it was decided to do separate tests in order to thoroughly investigate its impact on performance. While searching for parameter configurations it was assumed that for each test run the number of CUDA streams per device will be equal 128.

Table 1: Summary of the simulations results of the hybrid version for different computational tile sizes in the scenario involving 10 000 agents.

Scenario I								
Tile size		8x8x2	10x10x2	15x15x2	20x20x2	25x25x2	30x30x2	40x40x2
CPU threads		10	10	10	10	10	10	20
GPU devices		8	8	4	8	4	4	4
GPU tile threshold		15	5	3	3	3	5	15
Avg. sim. step duration [ms]	Avg	12.829	12.097	4.746	2.519	1.869	1.635	1.296
	Min	12.698	11.937	4.676	2.472	1.813	1.596	1.253
	Max	13.036	12.212	4.824	2.545	1.936	1.674	1.336
Tile processing share [%]	CPU	100.0	100.0	73.94	21.61	48.51	39.14	35.21
	GPU	0.00	0.00	26.06	78.39	51.50	60.86	64.79

Each test run assumes a simulation scenario lasting 60 seconds with a simulation step of 0.05s. The value of the radius of influence of the agents was set to 2m, which corresponds to the size of the margins of computational tiles.

The simulation environment used for testing is a rectangle of 2km by 1km and consists of 10 sectors with dimensions of 200m by 1km. Initially, agents are placed evenly on the left side of each sector. Their task is to move to the other end of the sector. Such a scenario might be useful for e.g. simulation of coordinated evacuation through a bottleneck part of a city, moving through a particular city district during a sports event etc.

Tables 1, 2, 3, 4 and 5 show simulation results for the best parameter configurations found for each simulation Scenario along with information on percentages of the computational tiles processed by the CPUs and by the GPUs. It is assumed that the main performance metric is the average time to perform a single simulation step. Final results are averages determined based on data from five runs.

Figure 7 shows that, in general, for different Scenarios, the optimal size of computational tiles (the one for which the best performance was obtained) is different. For more demanding Scenarios, tiles of small sizes perform best, while tiles of relatively large sizes perform best for less demanding Scenarios. This is primarily due to the varying density of agents on a tile for different Scenarios. If the density on a tile is high, it means that a single task delegated to a thread is demanding. In this case, as visible for Scenarios V, IV as well as III, reducing the size of the tile means reducing the complexity of the

Table 2: Summary of the simulations results of the hybrid version for different computational tile sizes in the scenario involving 50 000 agents.

Scenario II								
Tile size		8x8x2	10x10x2	15x15x2	20x20x2	25x25x2	30x30x2	40x40x2
CPU threads		10	10	10	20	20	20	20
GPU devices		8	4	4	4	4	4	4
GPU tile threshold		15	5	5	5	5	30	45
Avg. sim. step duration [ms]	Avg	22.136	10.511	5.807	4.123	3.847	3.450	3.247
	Min	21.732	10.425	5.664	4.049	3.807	3.410	3.232
	Max	22.597	10.589	5.994	4.282	3.922	3.494	3.262
Tile processing share [%]	CPU	100.0	58.74	40.05	42.88	23.96	19.45	11,51
	GPU	0.00	41.26	59.95	57.12	76.04	80.55	88.49

Table 3: Summary of the simulations results of the hybrid version for different computational tile sizes in the scenario involving 100 000 agents.

Scenario III								
Tile size		8x8x2	10x10x2	15x15x2	20x20x2	25x25x2	30x30x2	40x40x2
CPU threads		10	10	20	20	20	20	20
GPU devices		8	4	4	4	4	4	4
GPU tile threshold		10	10	25	10	40	40	25
Avg. sim. step duration [ms]	Avg	21.647	12.721	7.509	6.843	6.225	5.816	6.228
	Min	20.904	12.438	7.433	6.724	6.133	5.768	6.165
	Max	22.228	13.043	7.652	6.982	6.332	5.858	6.300
Tile processing share [%]	CPU	100.0	63.25	44.55	21.61	14.52	10.29	4.99
	GPU	0.00	36.75	55.45	78.39	85.48	89.71	95.01

Table 4: Summary of the simulations results of the hybrid version for different computational tile sizes in the scenario involving 500 000 agents.

Scenario IV								
Tile size		8x8x2	10x10x2	15x15x2	20x20x2	25x25x2	30x30x2	40x40x2
CPU threads		20	20	20	40	40	40	20
GPU devices		4	4	4	4	4	4	4
GPU tile threshold		25	25	15	15	15	75	25
Avg. sim. step duration [ms]	Avg	39.290	34.185	29.651	31.691	35.347	39.582	50.368
	Min	38.796	33.565	28.830	31.271	33.902	39.368	49.479
	Max	39.468	34.480	30.940	31.935	36.026	39.733	50.903
Tile processing share [%]	CPU	23.89	14.35	5.66	4.72	2.41	4.80	2.51
	GPU	76.11	85.65	94.34	95.28	97.59	95.20	97.49

Table 5: Summary of the simulations results of the hybrid version for different computational tile sizes in the scenario involving 1 000 000 agents.

Scenario V								
Tile size		8x8x2	10x10x2	15x15x2	20x20x2	25x25x2	30x30x2	40x40x2
CPU threads		20	20	40	40	40	10	10
GPU devices		4	4	4	4	4	8	8
GPU tile threshold		60	30	60	15	30	30	120
Avg. sim. step duration [ms]	Avg	67.886	63.569	67.228	78.811	92.946	109.404	138.823
	Min	66.012	61.492	66.341	76.545	91.028	108.846	138.245
	Max	68.952	65.119	68.373	80.808	93.777	110.098	139.291
Tile processing share [%]	CPU	12.27	4.95	4.84	1.77	2.65	0.50	1.32
	GPU	87.73	95.05	95.16	98.23	97.35	99.50	98.68

task, which proves to be beneficial in terms of better load balancing. On the other hand, as for smaller Scenarios I and II if the tile is too small, overheads become too large. Consequently, based on the observed results and the plots for the five scenarios we can generalize these findings into an optimization procedure that finds a minimum of a function observing the derivative over tile size approaching 0 and adopting the tile size at this, or close to this point.

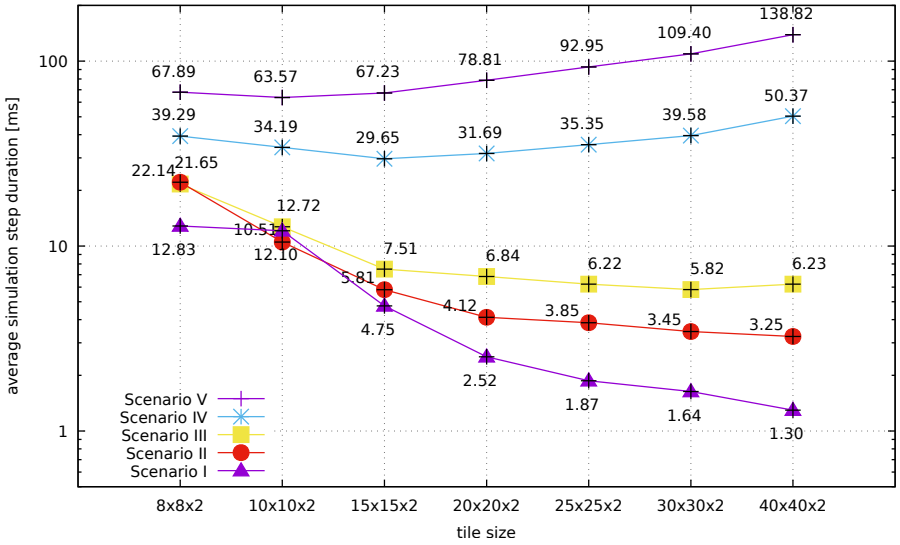

Fig. 7: Influence of the tile size on the simulation time.

Interestingly, in none of the Scenarios attempting to use all available

compute resources (i.e. 8 GPUs and 40 CPU cores) gave the best results. In fact, in every simulation the same configuration was the most optimal, namely 4 GPUs and 20 CPU threads, as shown in Figures 8 and 9. It appears that using too many resources may result in an increase in performance overhead due to more frequent/time-consuming synchronizations, the need to create more copies of the data, etc. which does not compensate for additional computing power.

The results also suggest that the best performance is observed when the vast majority of tiles were processed on the GPU side. It is particularly visible for Scenario V where, for the best configuration tested, about 95% of all computations were performed by GPUs.

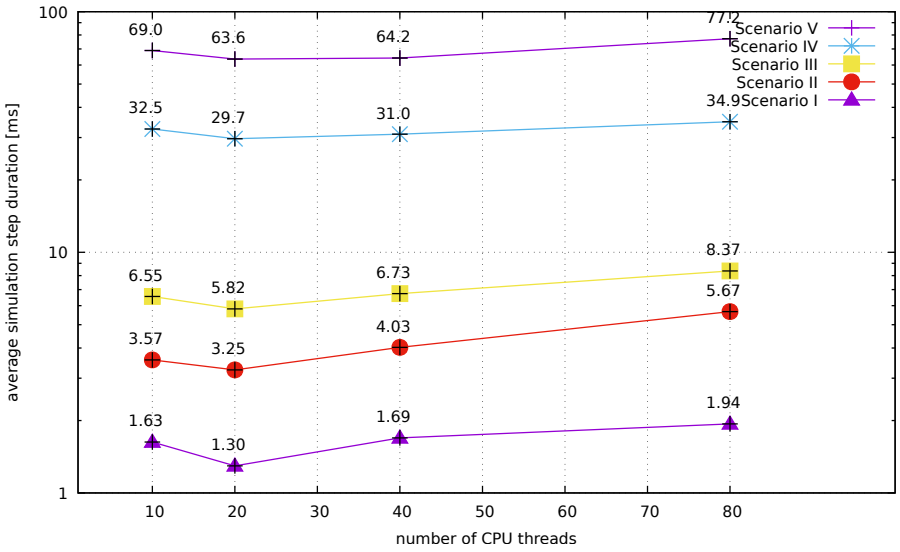

Fig. 8: Influence of the number of CPU threads used on the performance of the hybrid version of the solution

As mentioned before, using CUDA streams played a very significant role in terms of performance of the solution and for this reason it was tested separately. Figure 10 shows results of testing the impact of CUDA streams on the performance of the solution. For this test, the best parameter configuration for each Scenario was used; only the number of CUDA streams per device was adjusted. The graph in Figure 10 proves how important it was to use multiple CUDA streams in the implementation. In general, the more streams used the better the performance of the solution was. As previously

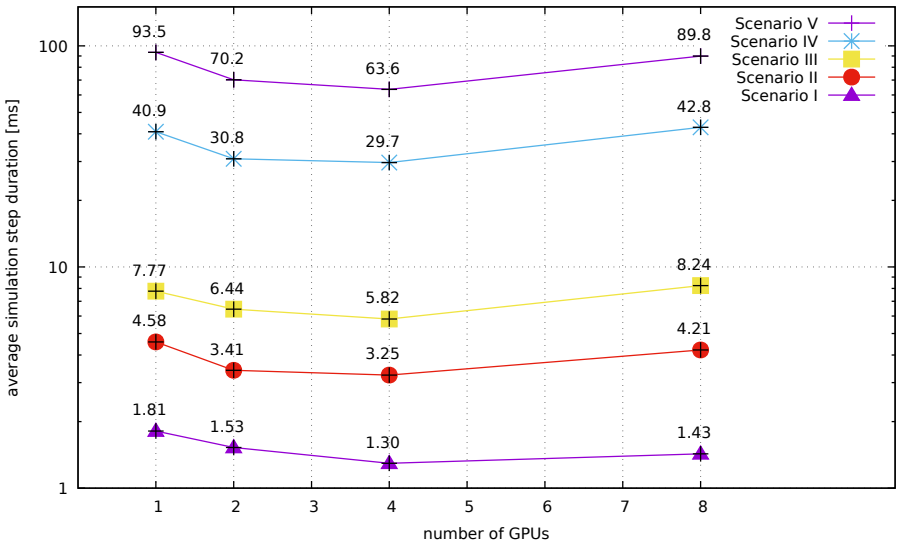

Fig. 9: Influence of the number of GPUs used on the performance of the hybrid version of the solution

assumed, the best performance was obtained for 128 streams per device. For this value the performance of the solution can be even more than 17 times higher (Scenario III) in comparison to the results obtained for only 1 CUDA stream per device. It shall be noted that the maximum number of resident grids per device for compute capability [28] of Quadro RTX 5000 used for tests is 128.

The reason why using CUDA streams brings such a large performance improvement comes from processing multiple computational tiles simultaneously. This behavior is shown in Figures 11, 12 and 13, which are screenshots from the NVIDIA Nsight Systems application that allows profiling of GPU performance. These figures show the results of profiling simulation scenarios using 8, 16 and 32 CUDA streams. It can be observed that the more streams used, the more kernels responsible for tile processing are executed in parallel.

Figure 14 shows how a single step of the main loop of the program is executed. All stages of the algorithm are distinguished in the profiler screenshot. From the GPU's perspective, Stage I focuses on transferring data from the host to the device and vice versa. As soon as these operations are complete, Stage II begins, which means that in this case, in Stage I the work on the CPU side has been fully occluded by GPU operations. Stage II consists

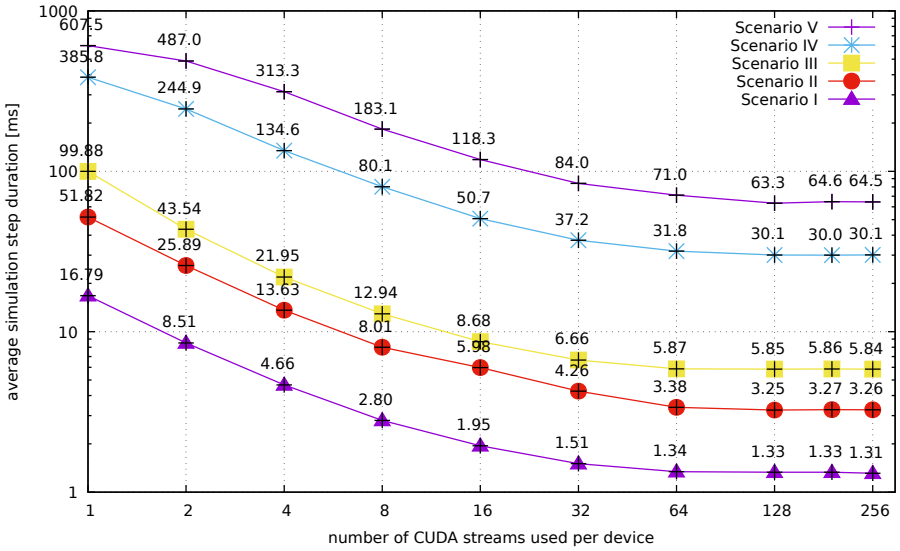

Fig. 10: Influence of the number of CUDA streams on the simulation time.

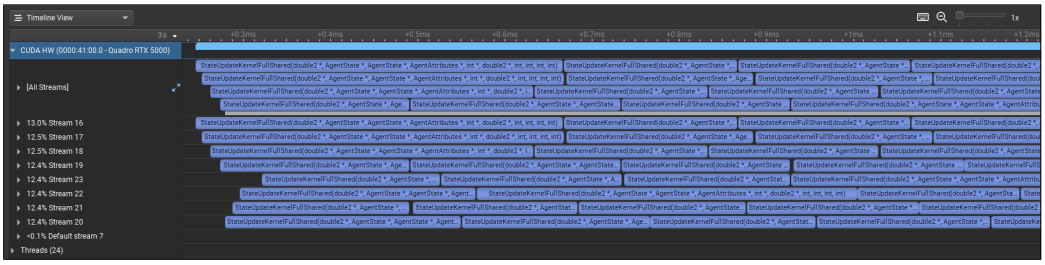

Fig. 11: NVIDIA Nsight Systems screenshot showing use of 8 CUDA streams.

primarily of processing the computational tiles. The workload is distributed evenly between all available devices and their CUDA streams. The work of the devices ends when the data is transferred to the host; however, this operation does not mark the end of this stage, as the processor threads can still continue their work. For this reason, the boundary between Stage II and III is marked with a dotted line. In Stage III, only the CPU performs work.

Using NVIDIA Nsight Systems for the 4 GPU configuration and the middle-sized Scenario III we have gathered statistics related to GPU kernel execution: registers per thread 42-64, dynamic shared memory 2912-6720 bytes, shared memory executed 65536 bytes, reported theoretical occupancy: 84.375%, 93.75% or 100%, during GPU involvement (communication and ker-

Fig. 12: NVIDIA Nsight Systems screenshot showing use of 16 CUDA streams.

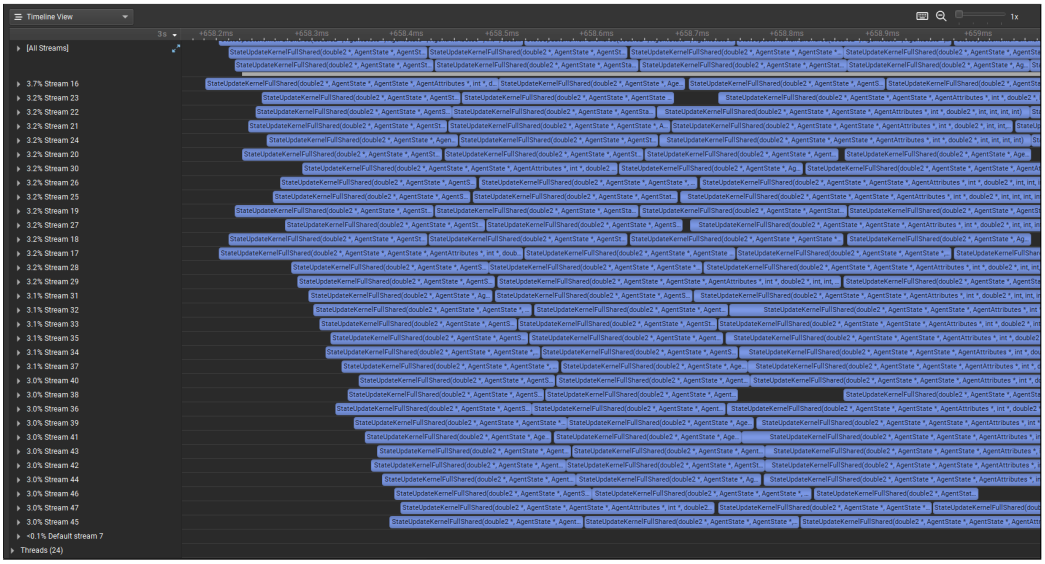

Fig. 13: NVIDIA Nsight Systems screenshot showing use of 32 CUDA streams.

nel execution) GPU active 100%, during the kernel execution phase SM active at the level of 70-80%, L2 read hit rate 30-100%, approx. 40% on average. In terms of processing of the most time-consuming operations: 89.2% was spent on kernel execution, 3.9% on synchronizing (cudaDeviceSynchronize()), 2.8% on HtD, 0.5% on DtH communication, 2.9% on kernel launching, 0.3% on global GPU memory allocation, 0.2% on pinned memory allocation.

Fig. 14: Screenshot from the profiler showing how a single simulation step is performed from a GPU perspective. All steps of the main loop algorithm are highlighted on the timelines.

4.2.2. Comparison with non-hybrid implementations

Comparison of the hybrid versus non-hybrid versions was made in order to determine if the extra programming effort put into developing a solution that takes advantage of CPU and GPU collaboration can actually bring significant performance benefits compared to implementations based on CPU or GPU usage only. Both non-hybrid implementations were built on top of the hybrid solution. The main challenge was to make changes so that one side (CPU or GPU) was able to determine new agent states independently. The overall approach to the problem remained the same.

The tests of both non-hybrid implementations focus primarily on finding the simulation configuration parameters that give the best results (similarly to the hybrid version) and comparing these results with the best corresponding outcome of the hybrid version. The tests were performed for the same simulation Scenarios, the same parameter values related to simulation time, and in the same simulation environment to eliminate the influence of these factors on the performance comparison. The method of testing was also the same as for the hybrid version which means that the final result for a given parameter configuration is the average of five measurements. The main performance measurement metric is the average time to execute a simulation step.

The test results of the non-hybrid implementations presented in Table 6 showed that in every Scenario considered, the hybrid version proved to be more efficient. For the most demanding Scenario a considerable speedup can

Table 6: Comparison of hybrid version performance with CPU-only and GPU-only versions performance for different simulation scenarios.

Hybrid vs CPU-only											
Best CPU configuration		**Scenario I**		**Scenario II**		**Scenario III**		**Scenario IV**		**Scenario V**	
Tile size		45x45x2		20x20x2		15x15x2		6x6x2		6x6x2	
CPU threads		20		20		20		20		40	
Results		**CPU**	**Hyb.**	**CPU**	**Hyb.**	**CPU**	**Hyb.**	**CPU**	**Hyb.**	**CPU**	**Hyb.**
Avg. sim. step duration	Avg	1.537	1.296	5.189	3.247	9.071	5.816	63.43	29.65	168.2	63.57
	Min	1.472	1.253	5.129	3.232	8.879	5.768	62.72	28.83	166.7	61.49
	Max	1.570	1.336	5.247	3.262	9.412	5.858	64.26	30.94	170.6	65.12
Avg. hybrid version speedup		1.186		1.598		1.560		2.139		2.646	
Hybrid vs GPU-only											
Best GPU configuration		**Scenario I**		**Scenario II**		**Scenario III**		**Scenario IV**		**Scenario V**	
Tile size		60x60x2		45x45x2		35x35x2		20x20x2		15x15x2	
GPU devices		4		2		2		2		2	
Results		**GPU**	**Hyb.**	**GPU**	**Hyb.**	**GPU**	**Hyb.**	**GPU**	**Hyb.**	**GPU**	**Hyb.**
Avg. sim. step duration	Avg	1.673	1.296	5.019	3.247	10.00	5.816	66.55	29.65	139.6	63.57
	Min	1.662	1.253	4.741	3.232	9.960	5.768	65.24	28.83	139.2	61.49
	Max	1.695	1.336	5.207	3.262	10.04	5.858	70.82	30.94	140.0	65.12
Avg. hybrid version speedup		1.291		1.546		1.719		2.244		2.197	

be observed. For Scenario V the hybrid version proved to be more than two times more efficient. The comparison of results in a graphical form is shown in Figure 15. The graph suggests that for scenarios with even more agents the advantage of the hybrid solution would be even greater.

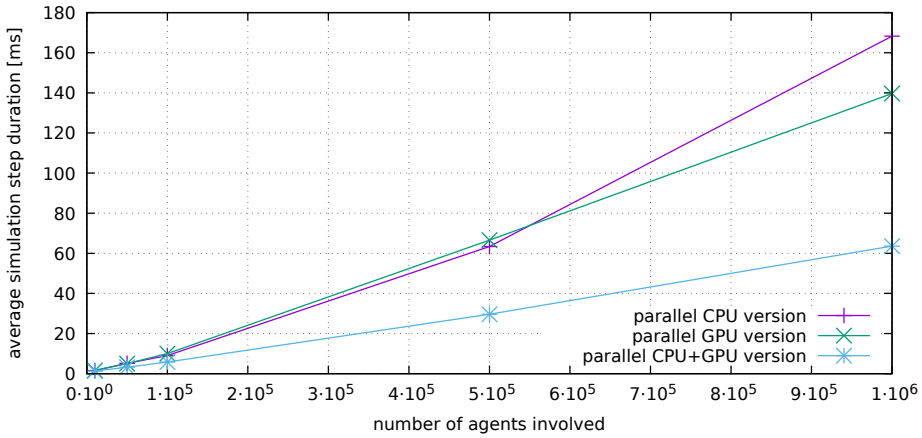

Fig. 15: Performance comparison of hybrid, CPU-only and GPU-only implementations. The plots include the results obtained for all Scenarios.

The inferior performance of the CPU-only version compared to the hybrid version is apparently due to the much smaller computational capabilities available. The CPU alone is not able to parallelize computations as massively without GPU(s)'s support. This means that far fewer computational tiles can be processed at the same time in the CPU version. It appears that not having to perform certain additional operations resulting from the presence of the GPU in the system (such as communication) does not compensate for the much smaller computing power available.

The performance differences between the hybrid version and the GPU version are due to less efficient data management on the CPU side. This is especially relevant for the third stage of the main simulation loop where all results are merged. In the hybrid version this stage is performed by all available threads, in the GPU version this task is performed only by the main thread, which is particularly problematic when the number of agents is very large.

Additionally, it can be observed that CPU-only implementation shows slightly better performance for least demanding simulation Scenarios than the GPU-only implementation. Specifically, up to the tested number of agents equal $5 \cdot 10^5$ the CPU version was marginally faster while the GPU one appeared significantly faster for 10^6 agents. This observation is in line with the fact [1] that GPU processing requires host-to-device and device-to-host communication that can be amortized by the smaller execution time compared to the CPU version only for problem sizes exceeding the given threshold which can depend on relative performance of the CPU and the GPU as well as the interconnect latency and bandwidth (PCIE in this case).

4.2.3. Comparison with other hybrid approaches

In [17] a CPU+GPU parallel method (OpenMP and 2D textures used respectively) has been shown to provide better performance than CPU and GPU only solutions by measurement of the simulation efficiency in fps, albeit limited in tests to small configurations i.e. AMD Athlon IIX3+NVIDIA GeForce GTS 250 i.e. up to 3 CPU threads and the GPU. For the number of characters in the range of 4 000 - 7 000 performance of the hybrid implementation was virtually the same as of the parallel CPU version and slightly better for 8 000 - 10 000 characters (approx. 51 vs 44 fps for the latter). In [4] authors presented an approach and corresponding results for scaling crowd simulations in a cluster composed of nodes each of which featured two CPU Intel Xeon E5649 6-Core CPUs, 24 GB RAM and two NVIDIA Tesla M2090 GPUs. Within each node 10 CPU cores and 2 GPUs were used for computations leaving 2 CPU cores for controlling GPUs. In terms of scenario sizes, configurations from 250 000 up to 126 million agents were tested, from 1 to 16 nodes showing largest speed-up for 16 nodes for the configuration with 4 million nodes of over 7. In terms of APIs, OmpSs, CUDA and MPI were used for an easy to program relatively high level task based approach using *#pragma omp target* directive. In [4] no scaling within one node versus CPU cores and GPUs is provided and the configurations within one node were limited to a relatively small configuration of 10 cores and 2 GPUs. Compared to these works, we contribute by proposing a way to parallelize among larger configurations tested up to 80 CPU threads and 8 GPUs, especially presenting how required partitioning changes depending on the CPU+GPU configuration. We test configurations between 10 000 and 1 000 000 agents and also show how using pinned and shared memories, data duplication and more advanced CUDA features such as CUDA streams and

running kernels in parallel contribute to scaling up to 4 GPUs and 40 CPU threads. Typically using APIs such as CUDA and proper compilers yield better performance than the level of offloading to GPUs from APIs such as OpenMP [29].

4.3. Discussion of the results

The tests of the hybrid implementation have shown that the solution is able to provide good computational performance, for each investigated Scenario. A prerequisite for satisfactory results is proper selection of simulation parameters. Based on the experimental results, the size of the computational tiles was found to have a significant impact on the performance. The size of the tiles that gives the best results depends primarily on the number of agents involved. In general, for a relatively small number of agents, the best performance was obtained for tiles of large size (40m x 40m), while for a relatively large number of agents, smaller tiles (10m x 10m) proved to be the most efficient. Interestingly, for each of the scenarios tested, using 20 CPU threads and 4 GPU devices to perform simulation calculations appeared to be the best from performance point of view. This configuration proved to provide the best balance between available processing power and the performance overhead resulting from the necessity of managing more computing resources. It was also observed that the use of CUDA streams had a significant impact on the performance of the solution. Involving 128 streams per device allowed to achieve at least 10 times (this value was different for various Scenarios) application speed-up.

A performance comparison of the non-hybrid implementations with the hybrid implementation proved that the hybrid variant is more efficient than both non-hybrid variants in every test case. This is especially noticeable for the most demanding Scenarios, involving a large number of agents. In such cases, the average execution time of a single simulation step is more than 2 times lower for the hybrid variant. This means that the extra programming effort required to develop an implementation that uses both the CPU and GPU for computation can be considered worthwhile. For complex and long-running simulation scenarios, this benefit should be particularly apparent.

One of the most challenging aspects of the hybrid implementation was the proper balance of workload between CPU and GPU. In our solution, we used the 'GPU tile threshold' parameter for this purpose (described in detail in Sections 3.5.1 and 4.2.1). It allows to decide which computational tiles should be processed on GPU and CPU. This approach is straightforward

and does not require too many additional calculations, but experiments have shown that its simplicity also has some drawbacks. For example, there is a high dependence between the size of the computational tiles and the 'GPU tile threshold', which means that every time the tile size changes, a new value of 'GPU tile threshold' has to be found in order to achieve the right workload balance. This makes it difficult to find the most optimal parameter configuration. One of the future challenges is to design a universal mechanism that is independent of simulation conditions and (other) simulation parameters.

5. Summary and future work

The solution developed within this paper, using a modern CPU+GPU system, allows to efficiently simulate crowd behavior. The implementation uses Helbing's model to describe the dynamics of the crowd. The application allows the use of a user-defined simulation environment, which makes it possible to create various simulation scenarios. We have shown for several scenarios of various sizes – the least demanding simulation scenario involved 10 thousand agents, and the most demanding involved 1 million agents – that our hybrid CPU+GPU implementation outperforms both CPU-only and GPU-only versions. For multithreading on the CPU side the OpenMP interface was used, whereas for handling GPU computation the CUDA platform was selected. The proposed solution is based on dynamic partitioning of the simulation environment into smaller regions called computational tiles. We have shown that the solution benefits from a number of optimizations to achieve the best possible performance including: load balancing among CPUs and GPUs, using multiple CUDA streams for parallel kernel execution, using pinned memory on the host and shared memory on GPUs.

Further work could focus on developing a mechanism to automatically select some of the simulation parameters, for example using the approach proposed in paper [30]. As tests have shown, the selection of appropriate parameter values is an important but also a challenging task. Furthermore, an attempt could be made to improve the algorithm for determining which tiles should be processed on the CPU and GPU. Currently this is done with a fixed threshold value which is one of the simulation parameters. Perhaps a slightly more sophisticated solution would contribute to a better balance of work between CPU and GPU. For example, the threshold value could be chosen dynamically based on data about the present and past positions of the agents.

[1] P. Czarnul, Parallel Programming for Modern High Performance Computing Systems, Chapman and Hall/CRC Press/Taylor & Francis, 2018.
URL https://www.crcpress.com/Parallel-Programming-for-\\Modern-High-Performance-Computing-Systems/Czarnul/p/book/9781138305953

[2] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, P. Dubey, Clearpath: Highly parallel collision avoidance for multi-agent simulation, in: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 177–187. doi:10.1145/1599470.1599494.
URL https://doi.org/10.1145/1599470.1599494

[3] E. B. Passos, M. Joselli, M. Zamith, E. W. G. Clua, A. Montenegro, A. Conci, B. Feijo, A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu, Comput. Entertain., 2010, 7 (4). doi:10.1145/1658866.1658879.
URL https://doi.org/10.1145/1658866.1658879

[4] H. Pérez, B. Hernández, I. Rudomín, E. Ayguadé, Scaling crowd simulations in a gpu accelerated cluster, in: I. Gitler, J. Klapp (Eds.), High Performance Computer Applications, Springer International Publishing, Cham, 2016, pp. 461–472.

[5] A. Malinowski, P. Czarnul, Multi-agent large-scale parallel crowd simulation with nvram-based distributed cache, Journal of Computational Science 33 (2019) 83–94. doi:https://doi.org/10.1016/j.jocs.2019.04.004.
URL https://www.sciencedirect.com/science/article/pii/S1877750318307221

[6] Y. Yu, D. Wen, Y. Zhang, X. Wang, W. Zhang, X. Lin, Efficient matrix factorization on heterogeneous cpu-gpu systems (2020). doi:10.48550/ARXIV.2006.15980.
URL https://arxiv.org/abs/2006.15980

[7] D. Yuanyuan, W. Xin, Application of cpu-gpu heterogeneous system in optical remote sensing image processing, Infrared and Laser Engineering

49 (11) (2020) 20200092.
URL https://www.researching.cn/articles/OJfb7a7c48882ba2cb

[8] P. Czarnul, Parallelization of large vector similarity computations in a hybrid CPU+GPU environment, J. Supercomput. 74 (2) (2018) 768–786. doi:10.1007/s11227-017-2159-7.
URL https://doi.org/10.1007/s11227-017-2159-7

[9] S. Yang, T. Li, X. Gong, B. Peng, J. Hu, A review on crowd simulation and modeling, Graphical Models 111 (2020) 101081. doi:https://doi.org/10.1016/j.gmod.2020.101081.
URL https://www.sciencedirect.com/science/article/pii/S1524070320300242

[10] W. van Toll, J. Pettré, Algorithms for microscopic crowd simulation: Advancements in the 2010s, Computer Graphics Forum 40 (2) (2021) 731–754. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142664, doi:https://doi.org/10.1111/cgf.142664.
URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142664

[11] S. R. Musse, V. J. Cassol, D. Thalmann, A history of crowd simulation: the past, evolution, and new perspectives, Vis. Comput. 37 (12) (2021) 3077–3092. doi:10.1007/s00371-021-02252-w.
URL https://doi.org/10.1007/s00371-021-02252-w

[12] J. Zhong, D. Li, Z. Huang, C. Lu, W. Cai, Data-driven crowd modeling techniques: A survey, (2022) ACM Trans. Model. Comput. Simul. 32 (1).

[13] E. Lobo-Hernández, X. Luo, G. Alomía-Peñafiel, N. Liu, C. Zúñiga-Cañón, How parallelization helps crowd simulation: Study of an openmp-based system, in: 2016 International Conference on Virtual Reality and Visualization (ICVRV), 2016, pp. 354–357. doi:10.1109/ICVRV.2016.66.

[14] X. Wang, Y. Zhang, D. Kong, B. Yin, A hybrid model for simulation of crowd evacuation, in: 2014 5th International Conference on Digital Home, 2014, pp. 347–352. doi:10.1109/ICDH.2014.72.

[15] M. Quinn, R. Metoyer, K. Hunter-Zaworski, Parallel implementation of the social forces model, in: Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, 2003.

[16] B. Hernandez, H. Pérez, I. Rudomin, S. Ruiz, O. de Gyves, L. Toledo, Simulating and visualizing real-time crowds on gpu clusters, Computación y Sistemas (2014) 651–664.

[17] X. Zhao, Y. Zhang, D. Kong, B. Yin, Comparision of real-time crowd simulation methods based on parallel architecture, in: 2012 Fourth International Conference on Digital Home, 2012, pp. 146–150. doi:10.1109/ICDH.2012.35.

[18] Opengl shading language, (February 2021).
URL https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

[19] D. Helbing, I. Farkas, T. Vicsek, Simulating dynamical features of escape panic, Nature 407 (6803) (2000) 487–490. doi:10.1038/35035023.
URL https://www.nature.com/articles/35035023

[20] L. Gibelli, N. Bellomo, Crowd Dynamics, Volume 1: Theory, Models, and Safety Problems, Birkhäuser Basel, 2018.

[21] M. Xu, C. Li, P. Lv, W. Chen, Z. Deng, B. Zhou, D. Manocha, Emotion-based crowd simulation model based on physical strength consumption for emergency scenarios, IEEE Transactions on Intelligent Transportation Systems 22 (11) (2021) 6977–6991. doi:10.1109/TITS.2020.3000607.

[22] F. M. Nasir, T. Noma, M. Oshita, K. Yamamoto, M. S. Sunar, S. Mohamad, Y. Honda, Simulating group formation and behaviour in dense crowd, VRCAI '16 Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry (2016) 289–292.

[23] T. D. Han, T. S. Abdelrahman, Reducing branch divergence in gpu programs, in: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, Association for Computing Machinery, New York, NY, USA, 2011. doi:10.1145/1964179.1964184.
URL https://doi.org/10.1145/1964179.1964184

[24] Y. L. J. W. J.-L. F. Yu Hao, Zhi-Jie Xu, A graphical simulator for modeling complex crowd behaviors, 2018 22nd International Conference Information Visualisation (2018) 6–11.

[25] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics 4 (2) (1968) 100–107. doi:10.1109/TSSC.1968.300136.

[26] B. Banerjee, A. Abukmail, L. Kraemer, Advancing the layered approach to agent-based crowd simulation, in: 2008 22nd Workshop on Principles of Advanced and Distributed Simulation, 2008, pp. 185–192. doi:10.1109/PADS.2008.13.

[27] P. Czarnul, Investigation of parallel data processing using hybrid high performance CPU + GPU systems and CUDA streams, Comput. Informatics 39 (3) (2020) 510–536.
URL http://www.cai.sk/ojs/index.php/cai/article/view/2020_3_510

[28] NVIDIA, CUDA C++ Programming Guide, v. 11.6, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (January 2022).

[29] E. Tiotto, B. Mahjour, W. Tsang, X. Xue, T. Islam, W. Chen, Openmp 4.5 compiler optimization for gpu offloading, IBM Journal of Research and Development 64 (3/4) (2020) 14:1–14:11. doi:10.1147/JRD.2019.2962428.

[30] P. Czarnul, P. Rościszewski, Auto-tuning methodology for configuration and application parameters of hybrid cpu + gpu parallel systems based on expert knowledge, in: 2019 International Conference on High Performance Computing & Simulation (HPCS), 2019, pp. 551–558. doi:10.1109/HPCS48598.2019.9188060.