Improving the GPU-based Collision Check Procedure for Distributed Crowd Simulations*

Guillermo Vigueras, Juan M. Orduña, Miguel Lozano
Departamento de Informática
Universidad de Valencia
Spain
juan.orduna@uv.es

José M. Cecilia, José M. García
Dpto. Ingenería y Tecnología de Computadores
Universidad de Murcia
Spain
{chema, jmgarcia}@ditec.um.es

ABSTRACT

The computing capabilites of current Graphics Processor Units (GPUs) have been used by many distributed applications for performing general purpose computations. In particular, the capabilites of many-core GPUs have been used in crowd simulations not only for enhancing the crowd rendering, but also for performing collision check and even for simulating the whole crowd. Nevertheless, these applications can still significantly increase their throughput if the GPU capabilities are fully exploited.

In this paper, we propose a new algorithm for GPU-based collision check in distributed crowd simulations. Unlike other collision check algorithms in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure, while keeping the crowd simulation consistent. The performance evaluation results show that the execution time required for the proposed method is significantly lower than previous methods based on sorting, increasing the crowd simulation throughput accordingly.

1. INTRODUCTION

The computing capabilites of current Graphics Processor Units (GPUs) have been used by many distributed applications for performing general purpose computations [11]. In particular, these capabilites have been used in crowd simulations, a special case of Virtual Environments where the avatars are autonomous agents instead of user-driven entities. Each of these agent-based entities can have its own goals, knowledge and behavior [14]. The computational cost of multiagent crowd simulations exponentially increases with the number of agents in the system, requiring a scalable design that can support huge amounts of agents (of different orders of magnitude) by simply adding more hardware. A distributed system architecture has been proposed to tackle these requirements [7, 17, 16]. That architecture consists of a distributed system where some of the computing nodes contain a distributed Action Server controlling the simulation. The rest of the computers host a set of agents implemented as threads of a single process. That architecture was shown efficient enough to support simulations up to tens of thousands of complex agents with plausible graphic quality. However, this distributed scheme can be still improved by fully exploiting the potential of new many-core architectures like GPUs.

Since the processing of the collision checks submitted by agents represents the most time consuming task in the distributed action server [17], in a previous work we implemented a basic distributed server for crowd simulations using an on-board GPU [18]. That GPU-based basic implementation used the particle algorithm [10] for performing parallel collision checks. Nevertheless, crowd simulations can still significantly increase their throughput if the GPU capabilities are fully exploited. In this paper, we propose a new GPU-based algorithm to perform the collision check procedure in distributed crowd simulations. Unlike other collision check procedures in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure, while keeping the consistency of the crowd simulation. The performance evaluation results show that the execution time required for the proposed method is significantly lower than previous methods based on sorting, increasing the system throughput accordingly.

The rest of the paper is organized as follows: Section 2 describes the distributed system for crowd simulation where the proposed GPU-based algorithm would be integrated. Section 3 briefly describes the related work about parallel architectures for crowd simulation. Section 4 gives an overview of the CUDA programming model. Next, Section 5 shows the proposed algorithm, as well as other improvements of existing methods for comparison purposes. Section 6 shows the performance evaluation results for the different approaches considered. Finally, section 7 shows some conclusion remarks.

*This work has been jointly supported by the Spanish MICINN and the European Commission FEDER funds under grants Consolider-Ingenio 2010 CSD2006-00046 and TIN2009-14475-C04.

2. A DISTRIBUTED SYSTEM FOR CROWD SIMULATION

In previous works, we proposed an architecture that can simulate large crowds of autonomous agents at interactive rates [7, 17, 16]. In that architecture, the crowd system is composed of many Client Computers, that host agents implemented as threads of a Client Process, and one Action Server (AS). The AS is executed in one computer and is responsible for checking the actions (eg. collision detection) sent by agents [7]. In order to avoid server bottleneck, the simulation world was partitioned into subregions and each one assigned to one parallel AS [17]. A scheme of this architecture is shown in Figure 1. This figure shows how the 2D virtual world occupied by agents (black dots) is partitioned into three subregions, and each one managed by one parallel AS (labeled in the figure as AS_x). Each AS is hosted by a different computer. Agents are execution threads of a Client Process (labeled in the figure as $Client_x$) that is hosted on one Client Computer. The computers hosting client and server processes are interconnected. Each AS process hosts a copy of the Semantic Database (SDB) that contains information about the simulated world. However, each AS exclusively manages the part of the database representing its region. In order to guarantee the consistency of the actions near the border of the different regions (see $agent_k$ in figure 1), the ASs can collect information about the surrounding regions by querying the servers managing the adjacent regions. Additionally, the associated Clients are notified about the changes produced by the agents located near the adjacent regions by the ASs managing those regions.

Each action requested by an agent requires a collision test in the corresponding AS. This test is computed based on the Area of Interest (AOI) of the agent. If the AOI of the considered agent does not intersect with the region border (eg. $agent_1$ in Figure 1), the corresponding AS updates the semantic database with the new location and notifies all the local CPs about that change. If, on the contrary, the AOI of the considered agent intersects with the region border (eg. $agent_k$ in Figure 1), then the adjacent servers are queried. Only if all the servers answer positively the requested action is allowed, and the semantic database is updated. In this case the queried adjacent servers are also notified about the change, in order to guarantee the consistency among all the SDB copies. The architecture shown in Figure 1 allows to simulate large crowds of autonomous agents providing a good scalability. However, this architecture can also benefit from the GPU capabilities for simultaneously checking the collision requests received from agents [18].

3. RELATED WORK

Some proposals have been made last year for exploiting the capabilities of multi-core and many-core architectures in crowd simulations. In this sense, a new approach has been presented for the CellBe processor to distribute the load among the processing elements [13]. Other work uses graphics hardware to simulate crowds of thousands of individuals using models designed for gaseous phenomena [2]. Recently, some authors have started to use GPU in an animation context (particle engine) [5], and there are also some proposals for running simple stochastic agent simulations on GPUs [8, 12]. However, these proposals are not suitable to simulate complex agents, including a cognitive model, at interactive rates.

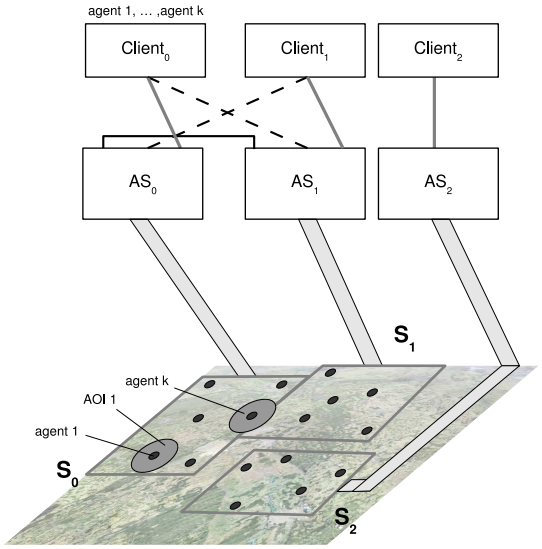

Figure 1: General scheme of the distributed architecture for crowd simulation

Other proposals show efficient GPU implementations of particle simulations [10] or parallel global pathfinding [1] using the CUDA programming environment. These works propose efficient models for a single GPU. On the contrary, this paper proposes a distributed implementation that can use as many GPUs as necessary, each one hosted by an Action Server, to perform the collision check process. In order to solve the GPU-based collision check problem, different implementations have been proposed [6, 20], based on hierarchical data structures and sorting. However, the computational cost of these proposals were shown efficient to solve problems like ray tracing but not for agent based simulation. Finally, another work proposes a GPU implementation for searching the k nearest neighbors in order to solve the collision check problem [4]. Nevertheless, this work does not assess the scalability of the method with the number of entities considered in the neighbors search.

4. CUDA PROGRAMMING MODEL

The Compute Unified Device Architecture (CUDA) programming model for GPU architectures covers both hardware and software features for performing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [9]. The hardware interface of CUDA consists of a parallel SIMD architecture, where thousands of threads run in parallel. These cores are organized as a given number of multiprocessors (SMs), each one having a set of 32-bit registers, constants and texture caches, and 16 KB of on-chip shared memory as fast as local registers (one cycle latency). At any given cycle, each core executes the same instruction on different data (SIMD), and communication among multiprocessors is performed through global memory.

CUDA consists of a set of C language library functions that the programmer can use to specify the structure of a CUDA

program. A CUDA program consists of two subprograms: The CPU (or *host*) subprogram and the GPU (or *device*) subprogram. The former prepares the execution on the GPU, moving data from main memory to the GPU memory, setting up all the parameters involved in the execution, and launching the code that is executed on the GPU by each thread.

The GPU subprogram consists of a set of kernels. Kernel execution is decomposed into blocks that run logically in parallel (they are physically executed only if there are resources available on the GPU). A block is a group of threads assembled by the developer which is mapped to a single multiprocessor. This group of threads can share 16 KB of memory and they can synchronize among them through barrier primitives. However, the communication among threads of different blocks is only performed through global memory, and the traditional way to synchronize them is terminating a kernel launch. All the threads are internally grouped into *warps*. A warp is a collection of threads that can run concurrently (with no time sharing) on all of the multiprocessors. The developer can determine the number of threads to be executed (up to a limit intrinsic to CUDA), but if there are more threads than the warp size, then they are time-shared on the actual hardware resources. Any thread can have access to all the GPU memory in the CUDA programming model, but there is a performance boost when threads access data located in shared memory, which is explicitly managed. Therefore, large data structures must be stored in the global memory and often-used data structures must be stored in the shared memory, in order to efficiently use the GPU's computational resources. This issue is particularly important in the collision check algorithms for crowd simulations.

5. COLLISION CHECK ALGORITHMS

The collision detection problem has been addressed in many areas like Computer Graphics, Computer Animation, Agent based Simulation, etc. A collision among agents within a crowd simulation occurs when the volume occupied by one agent intersects with the one occupied by other agent (this problem can be reduced to a two dimensional environment considering the 2D shape that represents each agent instead of its volume). Usually, the simulated scenario is divided by means of a n-dimensional grid in order to efficiently solve the collision check problem. In this way, only the agents contained in a given grid cell and the agents contained in the neighboring cells are checked. A naive GPU implementation of this grid (called *collision grid*) consists of defining a static array and assigning each grid cell to each position of this array. The mapping of agents to grid cells is performed by a spatial hashing method, depending on the cell size and the position of agents. Since many agents can fall within the same cell and GPU threads can simultaneously update the same memory address, atomic operations are needed in order to keep consistency [9]. However, atomic operations cause a performance penalty, increasing the execution time of the collision check procedure. Due to this penalty, other approaches based on sorting have been shown to obtain better performance than static approaches based on atomic operations [10, 3].

In a previous work, we implemented a collision check procedure for crowd simulations using an on-board GPU [18]. This algorithm consists of five steps, each one implemented as a GPU kernel. Figure 2 shows a scheme of the five steps and the data structures involved in this algorithm, as well as the input and output of each step. The upper part of Figure 2 shows a snapshot of a 2D grid, composed of sixteen cells containing six agents. In the lower part, this Figure shows the values of the data structures corresponding to that snapshot for each step of the algorithm. First, the hashing of the agents within the *collision grid* is performed, determining on which cell is located each agent (more than one agent can be assigned to the same cell). The result is an array containing the cell identifier assigned to each agent. Second, the sorting of the previous array based on the cell identifier (in increasing order) is performed, in order to allow the GPU threads to efficiently access to this grid. Third, the data structure containing the agents positions are also sorted to match the same cell order established in the second step. As a result, all the agents located in the same cell are in adjacent positions of the data structure. Fourth, a data structure representing the collision grid is computed. This structure allows a fast access to the agents located in each grid cell, and it consists of a sparse array. Finally, the last kernel is the collision check algorithm. This algorithm finds in which cell is located each agent and which other agents are located in the same or the neighboring cells (that is, the possible collisions are checked). In order to perform this task, it uses the data structure computed in the fourth step. The result is an array whose elements are a collision flag for each agent. We have denoted this algorithm as the *Baseline* implementation. Since this implementation is the basic translation of a GPU-based method for collision tests [10, 3], we have developed an improved implementation of that algorithm as a reference for comparison purposes.

5.1 Improved Baseline Algorithm

The *Baseline* algorithm is composed by five kernels. In order to improve the baseline algorithm, the first step is to determine which kernels are the most time consuming. We have measured the percentage of the global execution time consumed by each kernel for a simulation of one million agents. These measurements are shown in Figure 3. This figure shows that the most time consuming kernel is the one performing the collision check, consuming 63% of the total time. That is, this kernel does not take advantage of the GPU memory hierarchy in the *Baseline* version, since it only accesses the global memory.

Each agent checks its neighborhood in the collision check kernel. This data locality can be exploited by using the on-chip GPU memories. Concretely, the input arrays of the kernel performing the collision check can be bound to the texture memory. Hence, neighbor cells are cached and they can be fetched from the texture memory instead of the device memory, increasing the memory bandwidth. We have considered this as the first improvement of the baseline algorithm, and we have denoted it as the *texture memory* optimization. On other hand, data locality can be exploited by using the shared memory along with a tiling technique [19]. We define tiles within the collision grid in such a way that collisions can be independently checked by each GPU block, avoiding inter-block synchronization. We propose the ordering of the collision grid cells in global memory based on

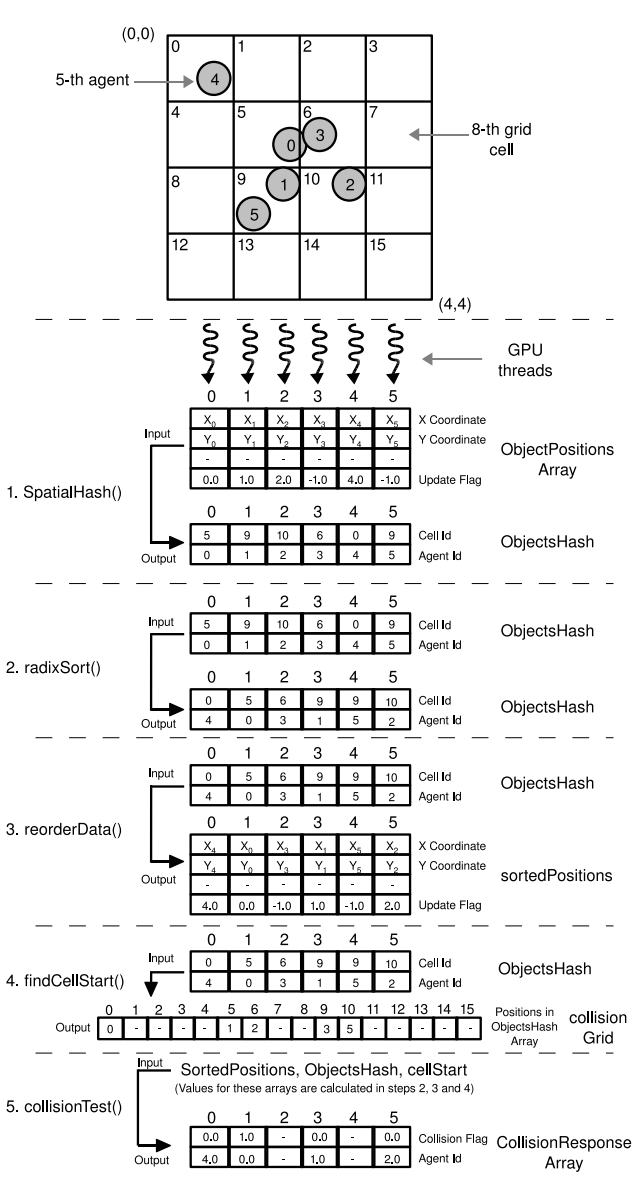

Figure 2: Baseline algorithm for GPU-based collision check

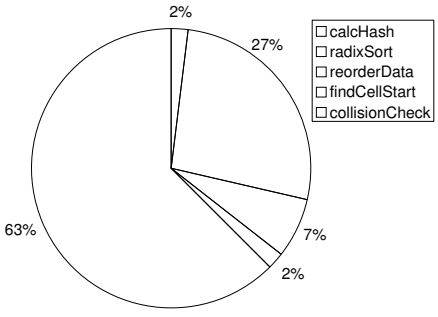

Figure 3: Percentage of execution time required by the kernels for the baseline version.

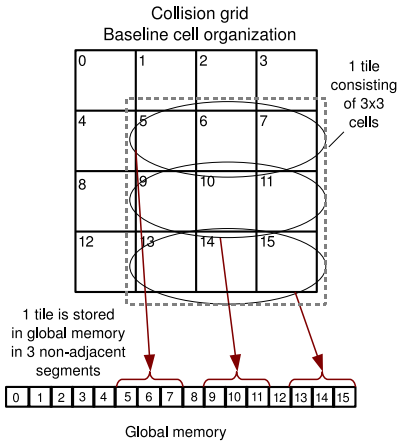

Figure 4: Grid mapping to global memory in the baseline version

the tile organization. In this way, all threads in a GPU block collaborate in loading the assigned tile from global memory to shared memory, obtaining a coalesced access and reducing the number of accesses to device memory. This memory layout also avoids bank conflicts in the access to shared memory. In order to illustrate this improvement, Figure 4 shows the memory access pattern of the baseline algorithm, while Figure 5 shows the memory access pattern of the improved baseline algorithm. Both figures show a collision grid with sixteen cells. Figure 4 shows how a given tile consisting of 3x3 cells (from cell 5 to cell 15 except cells 8 and 12) is stored in global memory. It can be seen that the neighboring cells are stored in non-adjacent memory segments (cells 8 and 12 are interleaved within the tile segments) preventing coalesced accesses to global memory.

A tile in the improved algorithm consists of 3x3 cells, as in the case of the baseline algorithm. Figure 5 shows how the improved algorithm replicates those cells that are in the border of a tile. In this figure, the numbers in the middle of each cell denotes the cell number in the collision grid, while the small numbers in the corners of each cell denote the replicas of that cell in each tile. For example, the cell number 3 is replicated as cell 4 in the first tile, cell 12 in the second tile, cell 19 in the third tile, and cell 27 in the fourth tile. The advantage of this data replication consists of having all the cells belonging to a given tile linearly ordered in the same global memory segment, as shown in the lower part of Figure 5. Therefore, all threads in a warp (half-warp) can linearly access to the same global memory segment and load the data into shared memory obtaining a coalesced access. We have denoted this improved organization along with the use of shared memory as the *shared memory* optimization. A key parameter in the shared memory optimization is the tile size, since it determines the number of threads in each block. In turn, this number of threads must be an entire multiple of the warp size in order to obtain a good performance. The tuning of the tile size should be experimentally performed. Concretely, a tile size of 16x16 (256 threads per block) has provided the best result for populations ranging from 10.000 to 1.000.000 agents.

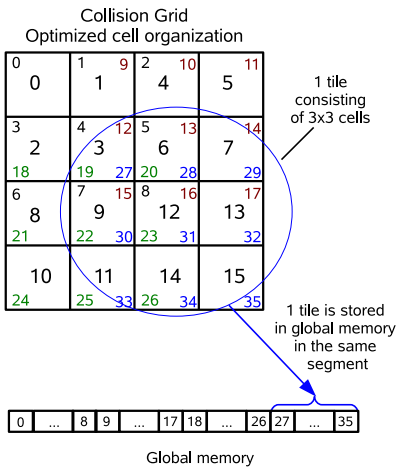

Figure 5: Grid mapping to global memory in the improved version

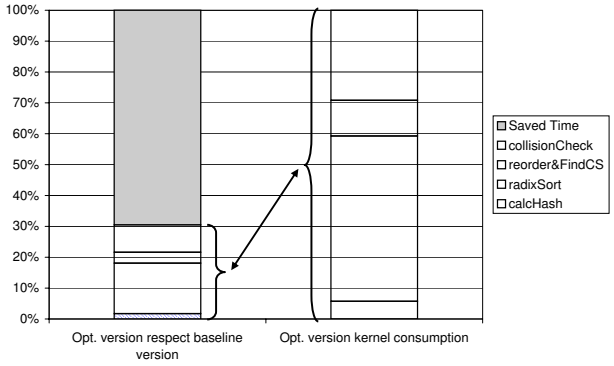

Figure 6: Percentage of execution times required by the kernels in the baseline optimized version.

Besides the collision check kernel, the kernel performing the radix sort is the second most time consuming kernel in the baseline algorithm (see Figure 3). In order to improve the performance, the radix sort procedure used in the baseline algorithm can be replaced by the fastest published version of this sorting algorithm [15]. Finally, although the execution time for the rest of the kernels are less significant than the previous ones, some optimizations can be performed on them. The kernels corresponding to the third and fourth steps in the *Baseline* algorithm can be merged into a single one, as there are no global synchronization requirements between them. Therefore, the cost of synchronization can be saved. Furthermore, the shared memory can be used by the fourth kernel, taking advantage of the data locality and improving the global memory bandwidth.

In order to show the improvements achieved by the optimized version of the Baseline algorithm, Figure 6 shows the impact of the optimizations in terms of percentages of the execution time (being 100% the total execution time of the Baseline algorithm on the left bar). This bar shows that the effect of the optimizations represents a reduction of a 70% in the global execution time respect to the Baseline version. The right bar in Figure 6 zooms in the results obtained for the improved version. In this version, the most time consuming kernel is the *radixSort*, with a 54% of the global execution time for the optimized version. For this reason, we propose a new algorithm to perform the collision check that is not based on sorting.

5.2 A New GPU-Based Algorithm for Collision Check

We propose an algorithm that avoids the sorting step in the collision check procedure. In order to achieve this goal, we use a static grid. Nevertheless, if many agents fall within the same grid cell and they try to write into the same memory address, atomic operations are needed. In order to avoid the performance penalty caused by atomic operations, we propose a different approach in which the size of each grid cell is fixed in such a way that the simulation consistency is guaranteed. Concretely, the consistency is guaranteed if

$$\sqrt{L^2 + L^2} = D = 2R \quad (1)$$

where L is the size of the side of a grid cell, D is the diagonal of a grid cell and R is the radius of the agents. When the distance between two agents is less or equal to twice the agent radius ($2R$) a collision occurs. For that reason the condition in Equation 1 establishes that all the agents falling in the same cell will collide since the maximum distance within a cell is the diagonal of the cell (i.e. $D = 2R$). In this way the condition in Equation 1 implicitly performs the collision detection for agents trying to move to the same cell. In that situation the consistency can be guaranteed by allowing the movement of one agent and forbidding the rest of the movements. It must be noticed that the selection of the agent to perform the movement can be done in a non-deterministic fashion since agent-based simulations evolve in this way.

As a result of using the condition in Equation 1 to define the cell size, more neighbor cells will have to be queried during the collision check. Since the side of a cell can be shorter than $2R$, not only the closest neighbor cells must be queried but also those cells that are one cell distant. We denote this set of cells as *extended neighbor cells*. Nevertheless, in spite of the higher number of neighbor cells accessed, the performance can be improved by loading these cells from global memory only once and store them on shared memory.

Using the consistency condition (equation 1), we have defined a new collision check algorithm consisting of four steps, each one containing one GPU kernel call. In this new algorithm there is an array (denoted as *CollisionResponseArray*) containing a pair *(collision flag, agent identifier)* in each position. Another array called *ObjectPositionsArray* contains the agents positions, and the array *collisionGrid* contains in each position three elements. The first element indicates the current step of the simulation. The second element stores an agent id indicating which is the target cell for that agent. The third element stores an agent id indicating which is the source cell for that agent. Agents positions are copied by the CPU onto device memory and then the collision check test is launched. Once the test is finished the result is returned back to the CPU by copying the *CollisionResponseArray*. The actions performed in each step of the new algorithm

are illustrated in Figure 7. This figure shows an example of the whole process, including the data structures involved as both input and output of each step. The upper part of this figure shows a snapshot of a 2D grid, composed of sixteen cells containing four agents at given locations. In the lower part, this figure shows the data structures with the values corresponding to that snapshot for each step of the algorithm described above. The actions performed in each step are the following ones:

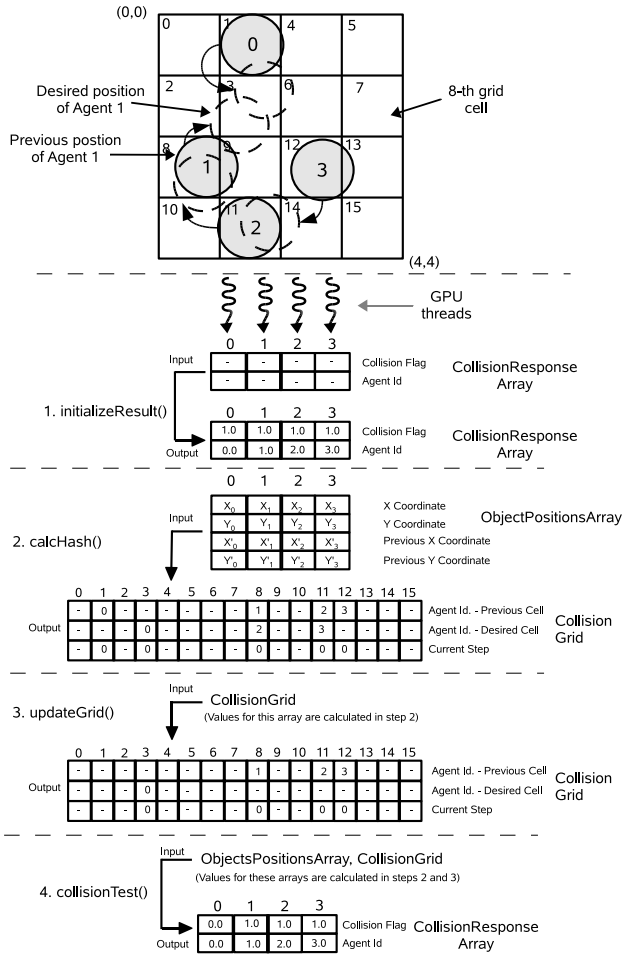

Figure 7: New algorithm for collisions check on the GPU

1. In the first step, the *collisionResponse* array is initialized indicating that there are collisions for all agents (see Figure 7). This initialization is necessary because one agent can overwrite other agent when falling in the same cell. Overwritten agents can detect the collision by means of this initialization step.

2. In the second step, the hashing to determine the target and the source cell for each agent position stored in *ObjectPositionsArray* is performed. Each thread writes a *step identifier* and the agent identifier in both the source and target cells. Since agents move at the same time in a simulation cycle of our tests, all the movements in the same cycle share a common *step identifier*. This identifier allows to determine whether the information within a cell is correct or it contains obsolete data. We use this *step identifier* to avoid using the function *cudaMemset( )*. The execution time of this function significantly increases the global execution time, specially when the size of the array to be cleared grows. The hashing performed in this step by the *calcHash* kernel is shown in Figure 7. Since cell 1 is the previous one for Agent 0 and it wants to move to cell 3, Agent 0 writes its identifier in these cells in the corresponding slot. Agent 2 moving from cell 11 to cell 8 and Agent 3 moving from cell 12 to cell 11, write their identifiers in the corresponding slots in these cells. Also Agent 1 writes its identifier in the proper slot of cell 8 (the source cell of Agent 1) but the value for the target cell (cell 3) is overwritten with the value stored by Agent 0 when the kernel *calcHash* finishes. All agents share the *step identifier* 0, since this is the first movement of each agent.

3. The third step of the new algorithm consists of agents detecting whether their desired movements are possible or not. If the desired movement of an agent was overwritten in the previous kernel or generates a collision, it means that the desired position is not possible. In this case, the collision grid is updated in the following way. Agents which desired movement was finally written, clean their identifier from their source cell. However if the movement of an agent is not possible it checks whether its source cell is the target cell of other agent. In such case the overwritten agent notifies that the desired movement is not possible. It must be noticed that restoring the previous position cannot lead to an inconsistent situation, since the initial scenario is collision free (i.e. position restore is possible), and for each cycle the agents positions are updated keeping the consistency. In Figure 7, Agent 0 cleans its identifier from its source position, cell 1. On the other hand, Agent 1 notifies to Agent 2 that its desired movement to cell 8 is not possible. Also Agent 2 notifies to Agent 3 that the desired position of the latter agent generates a collision.

4. Finally, the collision check is performed in the fourth step. For each grid cell, if the agent identifier stored in that cell is written in the *Desired Cell* slot then its *extended neighbor cells* are queried to detect a collision. If no collision is detected, then the collision flag in *collisionResponse* array is set to 0, indicating that there is no collision. If the movement is the previous one, then the collision flag is not overwritten, since the desired position generates a collision. Figure 7 shows that the collision for agent 1 is detected. The collision for agent 2 and agent 3 are also detected, since they are notified about it.

The algorithm described above performs global synchronization through finishing the second kernel launch. In this way, in the third kernel the overwritten agents are restored to their previous positions and the consistency of the simulation is kept. We have implemented a version of this algorithm using atomic operations for comparison purposes. This new version consists of merging the second and third steps in a single kernel. In order to merge these two steps,

atomic operations are needed (the global synchronization achieved through the second kernel termination should be performed by using atomic operations). However, the advantage of saving one kernel launch at the cost of using atomic operations should be analyzed.

6. PERFORMANCE EVALUATION

This section shows the performance evaluation of the GPU algorithms for collision check described in section 5. Our performance tests are based on different configurations of the simulated scenario, varying the number of agents, in order to evaluate the scalability of each algorithm version. We use random agent movements for evaluation purposes. Concretely, one hundred random movements are computed per agent, using the agent identifiers as the seed for the random generation, in order to obtain reproducible results. The execution times reported below are the aggregated time obtained for all the movements performed by all the agents considered for each simulation. Since the considered algorithm should scale up with the physical parallelism available on the GPU, we have considered different NVIDIA Tesla GPUs: the Tesla C870 (16 SMs) and Tesla C1060 (30 SMs).

Figure 8 and Figure 9 shows the overall execution time for the different collision check implementations on different graphic cards. These figures show on the X-axis the number of agents considered for the simulations. The Y-axis shows the aggregated execution time obtained for each collision check method. Figure 8 shows the results for the Tesla C870 platform. The new version using atomic operations has not been tested for this platform, since it does not support this kind of operations. As it could be expected, the greatest differences arise for the largest population size, that is, one million agents. We use the texture memory to decrease the use of the device memory in the first optimization, obtaining 50% of reduction in the execution time respect to the Baseline version. In the second optimization, the shared memory is used along with the new organization of the collision grid in global memory, in such a way that a coalesced access to device memory is guaranteed. This optimization obtains 70% of reduction in the execution time with respect to the Baseline version. Nevertheless, the proposed technique achieves the best results, obtaining 85% of reduction in the execution time.

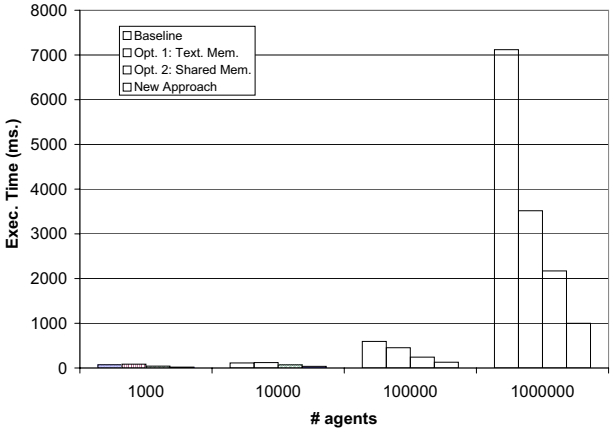

Figure 8: Execution times on Tesla C870 card

Figure 9 shows the execution times obtained for the Tesla C1060 card. In this case, the effects of the texture memory optimization hardly arise. The reason is that for this card the global memory access algorithm has been improved respect to the C870 platform [9], allowing to obtain more coalesced accesses. Therefore, the baseline algorithm requires much shorter execution times than for the case of the C870 card. The optimization that uses shared memory allows a decrease in the execution time of 53% with respect to the baseline version for a crowd size of one million agents. Nevertheless, the proposed algorithm achieves the best execution times, with a reduction of 65% when using atomic operations and around 75% without using atomic operations. If Figure 8 and Figure 9 are compared, then it can be seen that the execution times are inversely related to the number of SMs available on the cards.

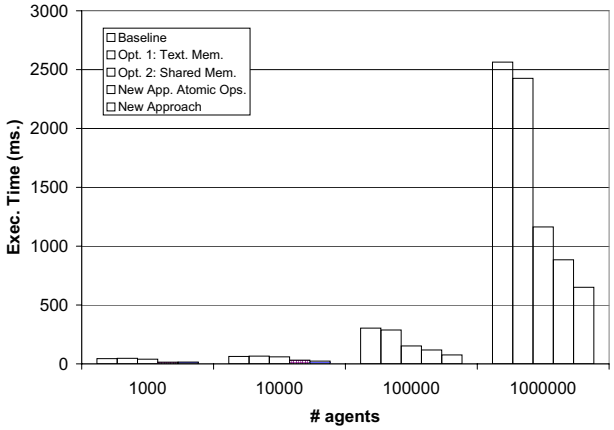

Figure 9: Execution times on Tesla C1060 card

In order to show that these execution times are directly related to the workload generated by each method, we have measured the throughput of the different versions in terms of number of collisions checked per second. Figures 10 and 11 show the collisions check rates obtained when increasing the number of agents for both the Tesla C870 and C1060 cards. Figure 10 and Figure 11 show that the proposed method without atomic operations performs the highest numbers of collision checks per second for all the population sizes. These figures also show that the collisions check rate performed by the proposed method significantly increases with the number of available SMs on the GPU, assessing the scalability of this method.

7. CONCLUSIONS

In this paper, we have proposed a new algorithm for GPU-based collision check in distributed crowd simulations. Unlike other collision check algorithms in the literature, the absence of both sorting procedures and atomic operations in the proposed method significantly reduces the computing workload of the collision check procedure while keeping the consistency of the crowd simulation. The performance evaluation results show that the execution times required for the proposed method are significantly lower than the ones of the methods used for comparison purposes, since the latter ones are based on sorting. Also, the results show that the number of collision checks per second achieved by the proposed method are the highest ones, showing that

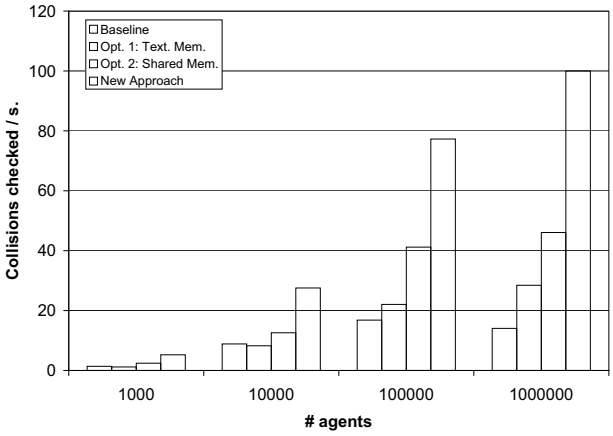

Figure 10: Collisions rate on Tesla C870 card

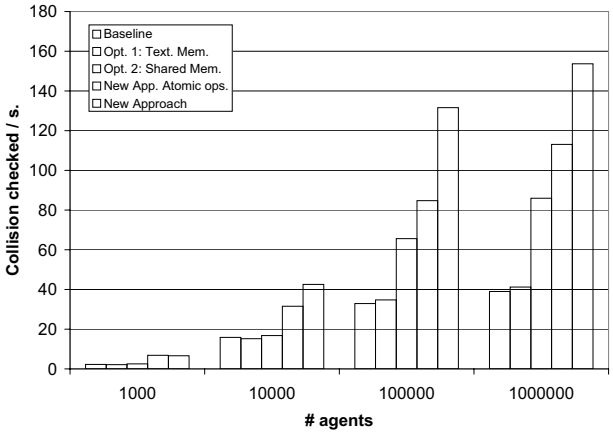

Figure 11: Collisions rate on Tesla C1060 card

the proposed method allows a higher throughput. Finally, the performance evaluation results show that the proposed method properly scales up with the number of multiprocessors available in the GPU.

8. REFERENCES

[1] A. Bleiweiss. Gpu accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, 2008.

[2] N. Courty and S. R. Musse. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, pages 206–212, 2005.

[3] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *Proceedings of HiBi 2009*, pages 51–58, Oct. 2009.

[4] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA, June 2008.

[5] L. Latta. Building a million particle system. In *In Proc. of Game Developers Conference(GDC-04)*, 2004.

[6] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum*, 28(2):375–384, 2009.

[7] M. Lozano, P. Morillo, J. M. Orduña, V. Cavero, and G. Vigueras. A new system architecture for crowd simulation. *J. Netw. Comput. Appl.*, 32(2):474–482, 2009.

[8] M. Lysenko and R. M. D'Souza. A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008.

[9] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[10] NVIDIA Corporation. *Particles Example. NVIDIA CUDA SDK*, 2008. Ver. 2.1.

[11] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[12] K. S. Perumalla and B. G. Aaby. Data parallel execution challenges and runtime performance of agent simulations on gpus. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 116–123, New York, NY, USA, 2008. ACM.

[13] C. Reynolds. Big fast crowds on ps3. In *Proceedings of the ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM.

[14] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.

[15] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of IEEE IPDPS '09*, pages 1–10, 2009.

[16] G. Vigueras, M. Lozano, J. M. Orduña, and F. Grimaldo. A comparative study of partitioning methods for crowd simulations. *Journal of Applied Soft Computing*, 10(1):225 – 235, 2010.

[17] G. Vigueras, M. Lozano, C. Perez, and J. Orduña. A scalable architecture for crowd simulation: Implementing a parallel action server. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP-08)*, pages 430–437, Sept. 2008.

[18] G. Vigueras, J. Orduña, and M. Lozano. *Advances in Practical Applications of Agents and Multiagent Systems*, chapter A GPU-Based Multi-Agent System for Real-Time Simulations, pages 15 – 25. Springer, April 2010.

[19] C. Xu, S. R. Kirk, and S. Jenkins. Tiling for performance tuning on different models of gpus. In *Proceedings of ISISE '09 : Int. Symp. on Information Science and Engineering*, 2009.

[20] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.