Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2014, Vol. 28(1) 33–49
© The Author(s) 2013
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342013476119
hpc.sagepub.com
SAGE

Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures

Guillermo Vigueras[1], Juan M Orduña[1], Miguel Lozano[1], José M Cecilia[2] and José M García[2]

Abstract

The computing capabilities of current multi-core and many-core architectures have been used in crowd simulations for both enhancing crowd rendering and simulating continuum crowds. However, improving the scalability of crowd simulation systems by exploiting the inherent parallelism of these architectures is still an open issue. In this paper, we propose different parallelization strategies for the collision check procedure that takes place in agent-based simulations. These strategies are designed for exploiting the parallelism in both multi-core and many-core architectures like graphic processing units (GPUs). As for the many-core implementations, we analyse the bottlenecks of a previous GPU version of the collision check algorithm, proposing a new GPU version that removes the bottlenecks detected. In order to fairly compare the GPU with the multi-core implementations, we propose a parallel CPU version that uses read–copy update (RCU), a new synchronization method which significantly improves performance. We perform a comparison study of these different implementations. On the one hand, the comparison study shows the first performance evaluation of RCU in a real user-space application with complex data structures. On the other hand, the comparison shows that the GPU greatly accelerates the collision test with respect to any other implementation optimized for multi-core CPUs. In addition, we analyse the efficiency of the different implementations taking into account the theoretical performance and power consumption of each platform. The evaluation results show that the GPU-based implementation consumes less energy and provides a minimum speedup of 45× with respect to any of the CPU-based implementations. Since interactivity is a hard constraint in crowd simulations, this acceleration of the collision check process represents a significant improvement in the overall system throughput and response time. Therefore, the simulations are significantly accelerated, and the system throughput and scalability are improved.

Keywords

Multi-core programming, GPU programming, crowd simulations, collision check procedure, performance improvement

1 Introduction

The computing capabilities of current multi-core and many-core architectures like graphic processing units (GPUs) have been used by many distributed applications for performing general purpose computations (Owens et al., 2007; Pratas et al., 2009; Goswami et al., 2010; Herault et al., 2010). One of the applications that can take advantage of the inherent parallelism of these architectures is crowd simulations. Crowd simulations are 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 (Reynolds, 1987). The computational cost of multi-agent crowd simulations increases greatly with the number of agents in the system, requiring a scalable design that can support huge numbers of agents (of different orders of magnitude) by simply adding more hardware.

In order to actually improve the scalability of crowd simulation systems, we proposed a distributed system architecture that can take advantage of the underlying distributed computer system (Vigueras et al., 2008; Lozano et al., 2009; Vigueras et al., 2010a, 2011). In addition, we implemented a preliminary version of a distributed server

[1] Departamento de Informática, Universidad de Valencia, Spain
[2] Departamento Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain

Corresponding author:
Juan Manuel Orduña, Unversidad de Valencia Avda de la Universidad, s/n Burjassot (Valencia), 46100 Spain.
Email: juan.orduna@uv.es

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

for crowd simulations using an on-board GPU (Vigueras et al., 2010b, 2010c).

In this paper, we propose different parallelization and code development strategies for the collision check procedure when executed on both multi-core and many-core architectures. In addition, we present a comparison study of the proposed implementations. We propose an implementation based on the traditional Mutex synchronization method and an implementation based on the read–copy update (RCU) synchronization mechanism (McKenney and Slingwine, 1998) as an optimization of the collision check procedure for multi-core architectures. In particular, the RCU-based implementation represents the first performance evaluation of RCU in a real user-space application with complex data structures. We also consider three different implementations of the collision check procedure on GPUs. The comparison study shows that the huge number of cores in the GPU is used to simultaneously validate movement requests from different agents, greatly reducing the execution time of the collision test with respect to any other implementation optimized for multi-core CPUs. As a result, one of the GPU-based implementations greatly reduces the execution time required for the collision check process with respect to the implementation of the same distributed server on a CPU, due to the absence of sorting procedures. We also analyse the efficiency of the different platforms by comparing the performance provided by the different implementations, taking into account the theoretical performance and power consumption of each platform. The performance evaluation results show that the GPU is the best platform since it consumes less energy and it provides a minimum speedup of 45× with respect to any of the CPU-based implementations. Since interactivity is a hard constraint in crowd simulations (Vigueras et al., 2008; Lozano et al., 2009), this reduction in the execution time of the collision test represents a significant improvement of the overall system throughput and response time. Therefore, the simulations are significantly accelerated, and the system throughput and scalability are improved. Also, the results show that the RCU mechanism better exploits the parallelism of a multi-core CPU than the most common techniques like Mutex.

The rest of the paper is organized as follows: Section 2 details the existing state-of-the-art of crowd simulations based on multi-core and many-core architectures; Section 3 describes two different implementations of the collision check procedure for multi-core processors; Section 4 describes the proposed optimizations of the GPU algorithm for performing the collision check procedure in crowd simulations; Section 5 demonstrates the performance evaluation of the different optimizations proposed and, finally, Section 6 contains our concluding remarks.

2 Related work

Recently, proposals have been made 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 (Reynolds, 2006). Other work uses graphics hardware to simulate crowds of thousands of individuals using models designed for gaseous phenomena (Courty and Musse, 2005). Recently, some authors have started to use GPU in an animation context (particle engine) (Latta, 2004; Peter et al., 2004), and there are also some proposals for running simple stochastic agent simulations on GPUs (Lysenko and D'Souza, 2008; Perumalla and Aaby, 2008). However, these proposals are not suitable for simulating complex agents, including a cognitive model, at interactive rates.

Other proposals show efficient GPU implementations of particle simulations (Par, 2008) or parallel global pathfinding (Bleiweiss, 2008) using the CUDA programming environment. These works propose efficient models for a single GPU. In contrast, this paper proposes a distributed implementation that can use as many GPUs as necessary to perform the collision check process.

The collision detection problem has been addressed in many areas, such as computer graphics, computer animation, agent-based simulation, etc. (Teschner et al., 2003; Guy et al., 2009). In the context of agent-based simulation, it consists of checking the collisions between agents that freely move within the same geometric space. A collision occurs when the volume occupied by one agent intersects with another agent (this problem can be reduced to a two-dimensional environment, by considering the two-dimensional shape that represents each agent instead of its volume). Although collision avoidance techniques have been developed for crowd simulations, they still fail to avoid some collisions when used in high-density environments (Guy et al., 2009). In order to efficiently solve the collision detection problem, usually the simulated scenario is divided by means of an n-dimensional grid (with n equal to 2 or 3 for two-dimensional and three-dimensional environments, respectively). In this way, only the agents contained in that grid cell and the agents contained in the neighboring cells are checked. A naïve implementation on the GPU of this grid (called the *collision grid*) consists of defining a static array and assigning each grid cell to each position of the array. The mapping of agents to grid cells is performed by the spatial hashing method, depending on the cell size and the position of agents. Since many agents can fall within the same cell, GPU threads can simultaneously update the same memory address. In order to guarantee the memory coherence of the application, atomic operations are needed for access to the same position in global memory by more than one thread (NVIDIA, 2010). Although atomic operations solve the coherence problem, they cause a penalty on performance, increasing the total execution time of the application (Par, 2008). Due to this penalty, other approaches based on sorting (Par, 2008) have been shown to obtain better performance than static approaches based on atomic operations. Also, there have been different implementations based on hierarchical

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

data structures and sorting for solving the GPU-based collision check procedure (Zhou et al., 2008; Kim et al., 2009; Lauterbach et al., 2009, 2010). However, the computational cost of these proposals was shown to be efficient for solving problems like ray-tracing but not for agent-based simulation.

Regarding collision avoidance techniques, it must be noted that all the different implementations of the collision test proposed in this paper should be executed on the action server (AS) of the distributed system for crowd simulation (Vigueras et al., 2008). However, the path-planning algorithm for animating agents within the crowd is executed in parallel on client orocesses (CP) being executed on different computers. Thus, the movements computed on the CPs are validated by the AS in order to keep the consistency of the simulation. Following this scheme, CPs can execute navigation algorithms like rule-based (Reynolds, 1987), social forces (Helbing et al., 2000) and reciprocal velocity obstacle (van den Berg et al., 2008) methods. However, these methods can fail to avoid collisions especially in high-density environments (Guy et al., 2009). For that reason, the proposed collision check implementations will be in charge of keeping the consistency, regardless of the navigation algorithms executed by the CPs.

Finally, we implemented a preliminary version of a distributed server for crowd simulations using an on-board GPU (Vigueras et al., 2010b). However, that was a single, preliminary version that was not optimized. In this paper, we go further, proposing different parallelization strategies and code development for the collision check procedure: parallel implementations using Mutex and RCU methods for the case of multi-core (CPU) architectures, and three different strategies for the case of many-core (GPU) architectures. In the latter case, we use the preliminary version shown in Vigueras et al. (2010b), as the baseline algorithm (see Section 4.1 below), and we improve this version in two different ways. We also present a comparison study of all the proposed implementations. In particular, the RCU-based implementation represents the first performance evaluation of RCU in a real user-space application with complex data structures. Additionally, one of the GPU implementations greatly reduces the execution time required for the collision check process with respect to the implementation of the same distributed server on a CPU, due to the absence of sorting procedures.

3 Accelerating the collision check procedure on multi-core CPUs

As multi-core processors become mainstream, multi-threaded applications will become more common, increasing the need for efficient programming models. Efficiently managing the data structures on a multi-core processor is relatively easy when the input of the parallel application is a static structure without data dependencies that can be partitioned between the execution threads. The OpenMP API for example, permits the parallelization of a program by means of directives, partitioning the workload between different execution threads. However, problems arise if dynamic data structures are not safely managed in a multi-threaded program. Furthermore, to obtain a proper speedup with the number of cores, an efficient thread coordination of concurrent accesses to shared data structures is needed. Traditional locking requires expensive atomic operations, such as compare-and-swap (CAS), even when locks are not contended. Locking is also susceptible to priority inversion, convoying, deadlock, and blocking due to thread failures. Therefore, many researchers recommend avoiding a locking-based synchronization. Some proposals use non-blocking (or lock-free) synchronization in multi-threaded applications for avoiding the use of locking, obtaining good results (McKenney and Slingwine, 1998; Sundell and Tsigas, 2008). Other work has studied the impact of replacing a locking-based synchronization with software transactional memory (STM) in a multi-player game server (Zyulkyarov et al., 2009; Gajinov et al., 2009). Nevertheless, these studies reveal that regardless the granularity of memory transactions used in STM, the performance obtained is even worse compared to a locking-based implementation.

This section describes two CPU implementations of the collision check problem using a grid data structure based on hashing. The first implementation is based on Mutex, the thread synchronization method provided by the POSIX threads API. The second implementation is based on a lock-free data structure, and the RCU method (McKenney and Slingwine, 1998) is used to protect this lock-free data structure from race conditions.

3.1 Mutex-based implementation of the collision check procedure

The Mutex version uses a grid-based data structure to perform the collision checking, this data structure is denoted as the *collision grid*. Figure 1 illustrates the implementation details of the Mutex-based version. The top part of the figure shows the geometric space partitioned using a grid with 16 grid cells. Four agents, represented as numbered circles, are allocated within the grid at given positions. The bottom part of Figure 1 shows that the collision grid is implemented as a lineal array. Each element of this array contains a Mutex and a pointer to a dynamic data structure, implemented as a linked list, that contains the agents positions. The Mutex avoids the corruption of the dynamic data structure protecting it when reader and writer threads access it concurrently during the collision check. A thread performs the collision check for an agent, calculating first the mapping of the agent's position into the collision grid by means of the hashing method. The Mutex located at the position returned by the hashing method is locked and the dynamic data structure is queried in the case of read access, or updated in the case of write access. The described implementation using an array of Mutexes instead of one Mutex protecting the collision grid, provides a certain level of

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

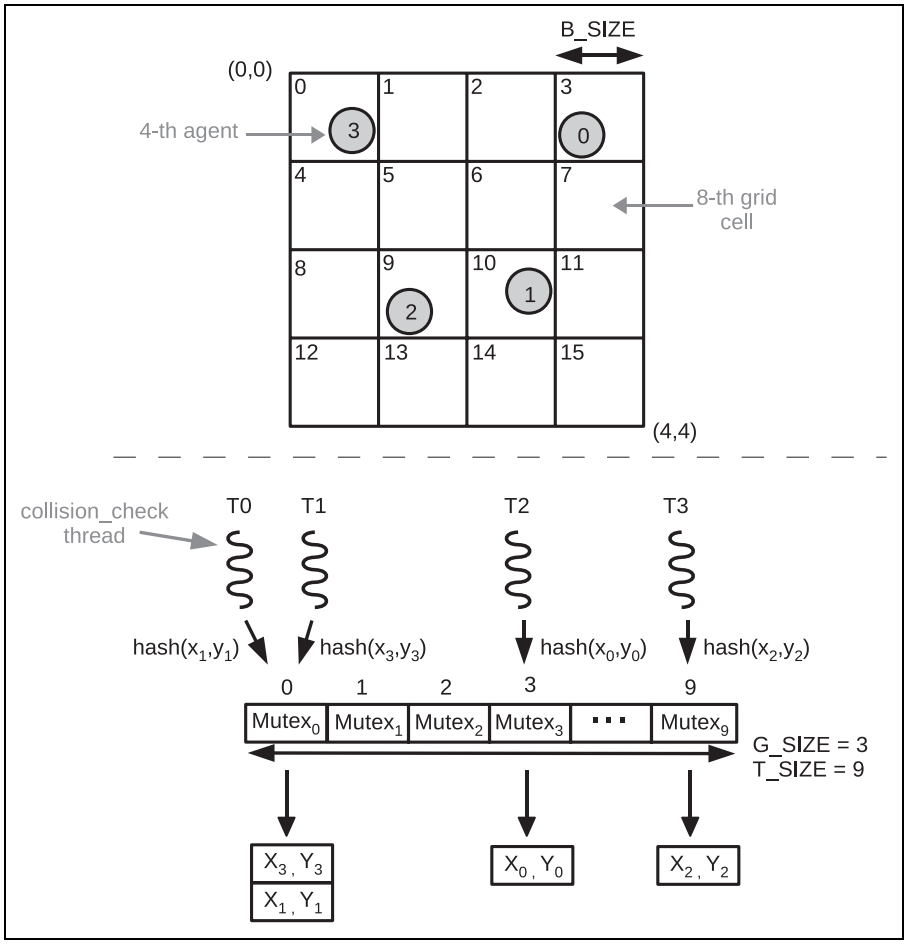

Figure 1. Diagram of CPU collision checking using Mutex.

concurrency between threads. However, the use of lock-free dynamic data structures along with the appropriate reclamation scheme can significantly improve the parallelism on a multi-core processor.

3.2 Read–copy update based implementation of the collision check procedure

Read–copy update (RCU) is a synchronization mechanism that was added to the Linux kernel during the development of version 2.5. In 2009 it was released for user-space access (Desnoyers, 2009). However, the benefits provided by RCU have not been evaluated in complex applications with large data structures. The idea behind RCU is to split updates into *removal* and *reclamation* phases. The removal phase removes references to data items within a data structure (possibly by replacing them with references to new versions of these data items), and can run concurrently with readers. The reason that it is safe to run the removal phase concurrently with readers is the fact that the semantics of modern CPUs guarantee that readers will see either the old or the new version of the data structure rather than a partially updated reference. The reclamation phase does the work of reclaiming (e.g. freeing) the data items removed from the data structure during the removal phase. Since reclaiming data items can disrupt any readers concurrently referencing those data items, the reclamation phase must not start until readers no longer hold references to those data items. Splitting the update into removal and reclamation phases permits the updater to perform the removal phase immediately, and to defer the reclamation phase until all readers active during the removal phase have completed, either by blocking until they finish or by registering a callback that is invoked after they finish. Only readers that are active during the removal phase need to be considered, because any reader starting after the removal phase will be unable to gain a reference to the removed data items, and therefore cannot be disrupted by the reclamation phase.

Different reclamation schemes can be used to implement RCU. We have implemented an RCU version using the quiescent state-based reclamation (QSBR) scheme since it provides concurrent reads with the lowest overhead but at the cost that the application has to be modified in order to explicitly manage reclamations (Hart et al., 2006; Desnoyers, 2009). QSBR uses the concept of a *grace period*. A *grace period* is a time interval $[a,b]$ such that, after time b, all nodes removed before time a may safely

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

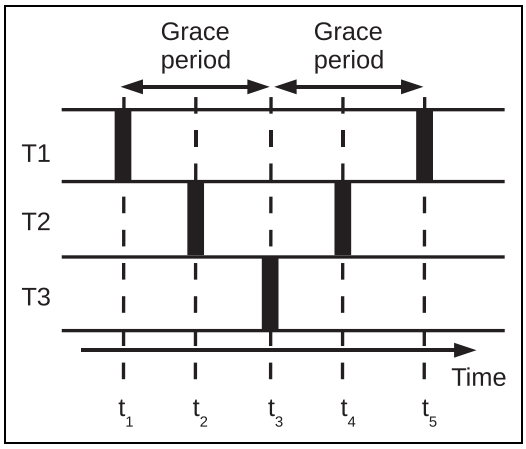

Figure 2. Illustration of QSBR. Black boxes represent quiescent states.

be reclaimed. QSBR uses quiescent states to detect grace periods. A quiescent state for thread T is a state in which T holds no references to shared nodes. Hence, a grace period for QSBR is any interval of time during which all threads pass through at least one quiescent state. Figure 2 illustrates the relationship between quiescent states and grace periods in QSBR. Thread $T1$ goes through quiescent states at times t_1 and t_5, $T2$ at times t_2 and t_4, and $T3$ at time t_3. Hence, a grace period is any time interval containing either $[t_1, t_3]$, or $[t_3, t_5]$.

Figure 3 shows the management of a linked-list in a multi-threaded environment by using RCU API calls (McKenney and Slingwine, 1998). The figure shows the changes suffered by a linked-list containing three elements (A, B and C) while an updater thread deletes element B. This element is deleted using the RCU API call *list_del_rcu ()*. This function removes a list element allowing some concurrent readers to continue seeing the removed element. Looking at Figure 3, it can be seen that after executing *list_del_rcu ()*, the element B has been removed from the list. Since readers do not synchronize directly with updaters, readers might be concurrently scanning this list. These concurrent readers might or might not see the newly removed element, depending on timing. However, readers that were delayed (e.g. due to interrupts) just after fetching a pointer to the newly removed element might see the old version of the list for quite some time after the removal. Therefore, there are two versions of the list, one with element B and one without it. The filling color of element B is still white during the grace period, indicating that readers might be referencing it, for that reason the freeing of element B is postponed.

Readers are not permitted to maintain references to element B after exiting from their RCU read-side critical sections. Therefore, when all readers have exited their critical sections, then no more readers can be referencing element B, as indicated by its grey color and dashed frame in the 'Updater' column in Figure 3. When no more readers hold references to element B, then the *synchronize_rcu ()* function completes. At this point, the list is back to a single version and element B may safely be freed.

We have adapted the CPU implementation based on Mutex described in Figure 1, in order to support lock-free data structures along with the RCU synchronization method. In this way, the lineal array representing the collision grid is modified removing the Mutex from each element of the collision grid. As a consequence, lock-free linked-lists containing the agents' positions are obtained. Read and update accesses to the lock-free linked-lists are performed by threads through a user-space RCU API (Desnoyers, 2009). This API allows us to manage the RCU removal phase defining read-side critical sections in order to run write accesses to shared linked-lists concurrently with read accesses. The reclamation phase is managed using the QSBR reclamation scheme within RCU. The QSBR version of RCU implemented by this API, permits the definition of quiescent states in order to safely update or reclaim the linked lists nodes contained in the collision grid.

4 Optimization of the collision check procedure on graphic processing units

All NVIDIA GPU platforms from the G80 architecture can be programmed using the compute unified device architecture (CUDA) programming model which makes the GPU operate as a highly parallel computing device (NVIDIA, 2010). Each GPU device is a scalable processor array consisting of a set of single instruction multiple threads (SIMT) streaming multi-processors (SM), each of them containing several stream processors (SPs). Different memory spaces are available in each GPU on the system. The global memory (also called the *device* or *video memory*) is the only space accessible by all multi-processors. It is the largest and the slowest memory space and it is private to each GPU on the system. Moreover, each multi-processor has its own private memory space called its *shared memory*. The shared memory is smaller and also provides lower access latencies than global memory. Finally, there are other addressing spaces for specific purposes such as texture and constant memory (NVIDIA, 2010).

The CUDA programming model is based on a hierarchy of abstraction layers. The *thread* is the basic execution unit that is mapped to a single SP. A *thread-block* is a batch of threads which can cooperate together as they are assigned to the same multi-processor, and therefore they share all the resources included in this multi-processor, such as register file and shared memory. A *grid* is composed of several thread-blocks which are equally distributed and scheduled among all multi-processors. As an aside, there is no particular order to the way the thread-blocks are executed, therefore they are executed in multiple instruction multiple data (MIMD) fashion. Finally, threads included in a thread-block are divided into batches of 32 threads called *warps*. The warp is the scheduled unit, so the threads of the same thread-block are scheduled in a given multi-processor warp

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

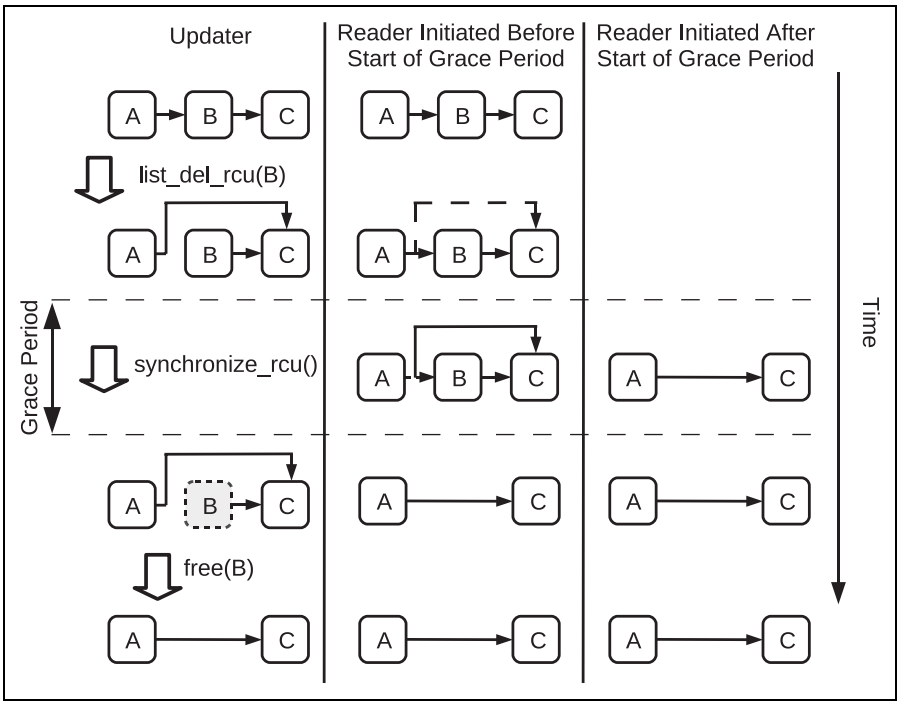

Figure 3. Deletion of one element in an RCU linked-list.

by warp. The 32 threads in a warp execute the same instruction over multiple data (SIMD). The programmer declares the number of thread-blocks, the number of threads per thread-block and their distribution to arrange parallelism given the program constraints (i.e. data and control dependencies).

This section describes a new GPU algorithm, implemented with CUDA (NVIDIA, 2010), for performing the collision check procedure. Additionally, in order to make this paper self-contained, this section briefly describes the preliminary version of the algorithm for the GPU-based collision check procedure proposed in our previous work (Vigueras et al., 2010b), denoted as the *baseline version*. This algorithm has been used in other works about GPU-based collision detection and particle simulation (Par, 2008; Erra et al., 2009), therefore becoming a reference for comparison with other work. Also, several optimizations of the baseline algorithm (Vigueras et al., 2010c) are described, for comparison purposes.

4.1 Baseline algorithm

The baseline algorithm uses a grid to perform the collision test (Vigueras et al., 2010b). The dimensions of this grid, the grid cell size and grid origin coordinates are input parameters that depend on the simulated scene.

The algorithm consists of five main steps, each one represented by one CUDA kernel: First, the *spatialHash* kernel updates the collision grid, performing the spatial hashing by means of the agents' positions given as inputs. Second, a *radixSort* (Harada et al., 2007) is performed for ordering the output of the previous step based on the cell identifier (the first being the lowest cell identifier). This sorting is needed to allow efficient access to global memory during the collision check step. Third, the coordinates of the agents given as inputs are sorted based on the order established in the previous step. Fourth, the data contained in the collision grid is indexed in order to allow a quick access to the agents in neighboring cells. Finally, the collision test is performed.

4.2 Improved baseline algorithm

The baseline algorithm is composed of five kernels. In order to improve the baseline algorithm, the first step is to determine which kernels are the most time-consuming. The percentage of the global execution time consumed by each kernel was measured when checking the movements of 1,000,000 agents. These measurements are shown in Figure 4. This figure shows that the most time-consuming kernel is the one performing the collision check, consuming 63% of the total time. The main reason for this time consumption is that this kernel does not take advantage of the GPU memory hierarchy in the baseline version, accessing only to global memory.

During the collision check kernel (the fifth kernel in the baseline algorithm), each agent checks its neighborhood. This data locality can be exploited by using the on-chip GPU memories. 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. This first improvement of the baseline algorithm is denoted the *texture memory*

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

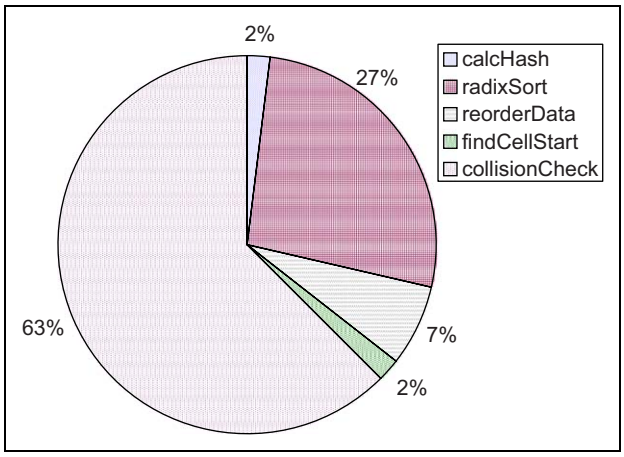

Figure 4. Percentage of execution time required by the kernels for the baseline version.

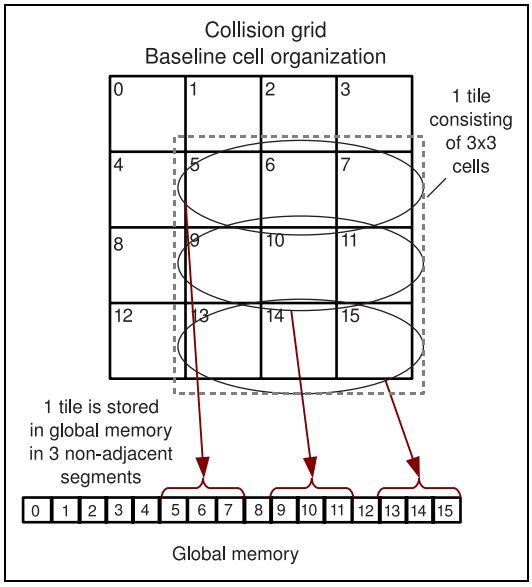

Figure 5. Grid mapping to global memory in the baseline version.

optimization. On the other hand, data locality can be also exploited by using the shared memory along with a tiling technique (Xu et al., 2009). Tiles are defined within the collision grid in such a way that collisions can be independently checked by each GPU block, avoiding inter-block synchronization. Collision grid cells are ordered in global memory based on 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 5 shows the memory access pattern of the baseline algorithm, while Figure 6 shows the memory access pattern of the improved baseline algorithm. Both figures show a collision grid with sixteen cells. Figure 5 shows how a given tile consisting of 3×3 cells (from cell 5 to cell 15, except cells 8 and 12) is stored in global memory.

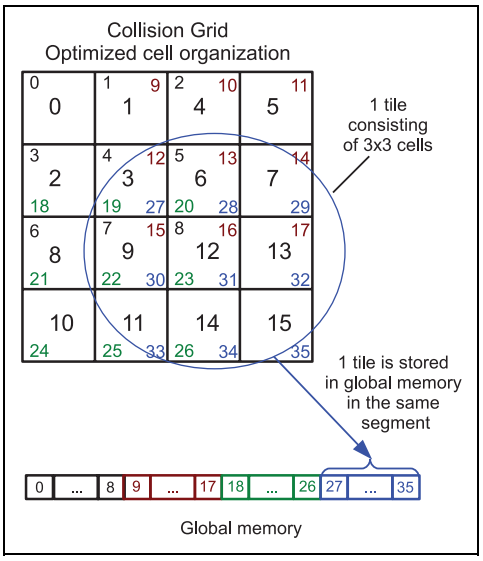

Figure 6. Grid mapping to global memory in the improved version.

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.

Figure 6 shows the global memory layout for the improved version. A tile in the improved algorithm consists of 3×3 sub-matrix of cells, as in the case of the baseline algorithm. This 3×3 sub-matrix is composed of a 2×2 sub-matrix of cells and its neighbor cells. For example, the tile highlighted in Figure 6 with a blue circle, consists of cells 12, 13, 14 and 15 forming the 2×2 sub-matrix that along with cells 3, 6, 7, 9 and 11 form the 3×3 tile. For that reason, cell numbers (i.e. big numbers in the middle of each cell) are assigned in the improved version for keeping those cells contained in the 2×2 sub-matrix linearly ordered. In addition, the improved algorithm replicates those cells that are in the border of the 2×2 sub-matrix of a tile. Figure 6 shows this replication scheme. In this figure, the numbers in the middle of each cell denote 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. Therefore, all threads in a warp (half-warp) can linearly access the same global memory segment and load the data into shared memory obtaining a coalesced access. The lower part of Figure 6 shows how the cells of the first tile (black numbers in the corner of each cell) are stored in the same global memory segment. The same occurs for the second tile (numbers in red), for the third tile (numbers in green) and for the fourth tile (numbers in blue). This improved organization along with the use of shared memory is denoted as the *shared memory* optimization. It should be

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

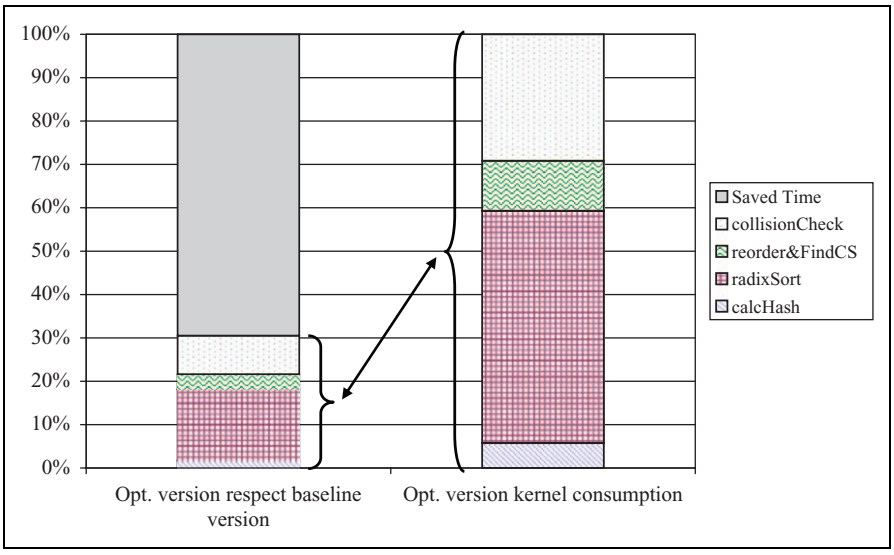

Figure 7. Percentage of execution times required by the kernels in the baseline optimized version.

noticed that GPU memory consumption is not a hard limitation since crowd simulations are performed in a distributed fashion. In this way, the replication of border cells cannot exhaust GPU memory, since the simulated world can be distributed across several GPU-based action servers (Vigueras et al., 2008, 2010b).

Figure 4 shows that the collision check is the most time-consuming kernel in the baseline algorithm. However, this figure also shows that the kernel performing the *radixSort* requires 27% of the total execution time, being the second most time-consuming kernel. Since the *radixSort* is the fastest GPU sorting method (Satish et al., 2009), the *radixSort* procedure used in the baseline algorithm has been replaced by the fastest published version of this sorting algorithm (Satish et al., 2009). Finally, although the execution times 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 7 shows the impact of the optimizations in terms of percentages of the execution time (100% being 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 70% in the global execution time with respect to the baseline version. The right bar in Figure 7 zooms into the results obtained for the improved version. In this version, the most time-consuming kernel is the *radixSort*, using 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.

4.3 New graphic processing unit based algorithm for collision check

This section describes a new algorithm for performing the GPU collision check. This algorithm avoids the sorting step in the collision check procedure. In order to achieve this goal, a static grid is used. 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, a different approach is proposed in which the size of each grid cell is fixed in order to guarantee the consistency of the simulation. The consistency is guaranteed if

$$\sqrt{L^2 + L^2} = D = 2R \tag{1}$$

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 noted 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. This set of cells is denoted as *extended neighbor cells*. Nevertheless, in spite of the fact that more neighbor cells

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

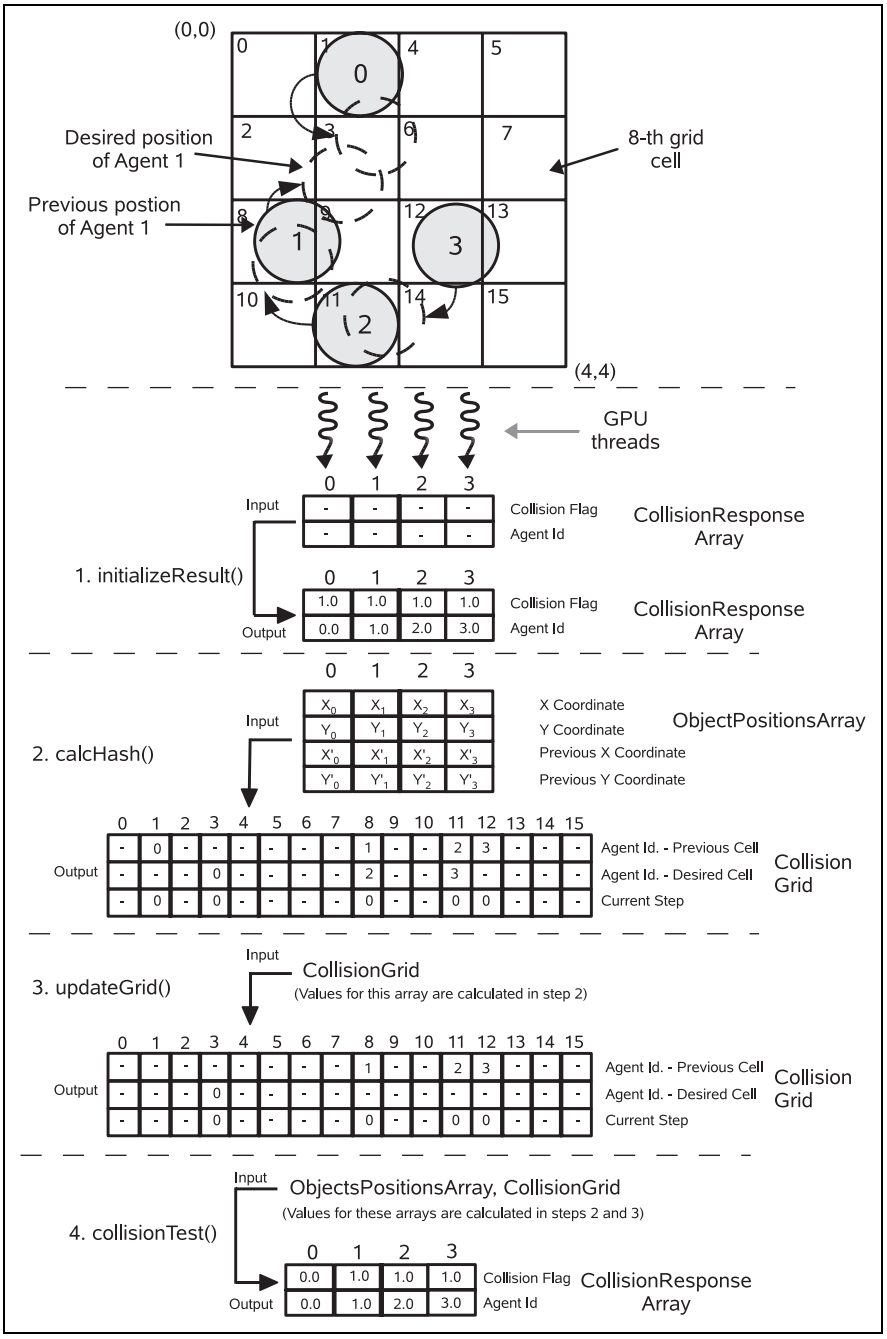

Figure 8. New algorithm for collision check on the GPU.

are accessed, the performance can be improved by loading these cells from global memory only once and storing them on shared memory.

Using the consistency condition (equation (1)), a new collision check algorithm has been defined consisting of four steps, each one containing one GPU kernel call. In this new algorithm there is an array (denoted as *CollisionResponseArray*) containing each element a pair *(collision flag, agent identifier)*. Another array called *ObjectPositionsArray* contains the agents positions, and the array *collisionGrid* has as many positions as cells used to perform the collision check. In addition, the *collisionGrid* array contains three elements in each position. The first element indicates the current step of the simulation. The second element in a given position *i*, stores an agent identifier indicating that the target cell for that agent is cell *i*. The third element in a given position *i*, stores an agent identifier indicating that the source cell for that agent is cell *i*.

Before the collision check test is launched, agents' positions are copied by the CPU onto device memory. Once the test is finished the result is returned back to the CPU by copying the *CollisionResponseArray*. Actions performed in each step of the new algorithm are illustrated in Figure 8. This figure shows an example of the whole process,

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

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:

In the first step, the *collisionResponse* array is initialized indicating that there are collisions for all agents (see Figure 8). This initialization is necessary because one agent can overwrite another agent when falling in the same cell. Overwritten agents can detect the collision by means of this initialization step.

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. All updated positions share a common *step identifier*. This identifier allows us to determine whether the information within a cell is correct or if it contains obsolete data. This *step identifier* is used to avoid using the function *cudaMemset ( )* for clearing the content of *collisionGrid* before launching the second step. The execution time of this function significantly increases the global execution time, especially when the size of the array to be cleared grows. The hashing performed in this step by the *calcHash* kernel is shown in Figure 8. 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. In addition, 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 movements for these agents are checked in the same collision test launch.

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 that case, the collision grid is updated in the following way: agents whose desired movement was finally written clean their identifier from their source cell. However, if an agent detects that its movement is not possible, it checks whether its source cell is the target cell of other agent. In such cases, the overwritten agent notifies that the desired movement is not possible. It must be noted 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 8, Agent 0 cleans its identifier from its source position, cell 1. On the other hand, Agent 1 notifies Agent 2 that its desired movement to cell 8 is not possible. In addition, Agent 2 notifies Agent 3 that the desired position of the latter agent generates a collision.

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. On the other hand, if the agent identifier is written in the *Previous Cell* slot, then the collision flag is not overwritten, since the desired position for that agent generates a collision. Figure 8 shows that the collision for Agent 1 is detected. The collisions 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. A version of this algorithm has been implemented 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.

5 Performance analysis

This section shows the performance evaluation of both CPU and GPU algorithms of the collision check procedure described in Sections 3 and 4. Our performance tests are based on different configurations of the simulated scenario, increasing the number of agents, in order to evaluate the scalability of each algorithm version. Agents are uniformly distributed and the size of the scene is proportionally increased in order to obtain low-density scenarios. The collision test algorithms described before iterate over the set of neighbor cells until any obstacle is found. For that reason, low-density represents the worst case scenario (the absence of surrounding obstacles results in more neighbor cells checked during the collision test (Lozano et al., 2009)). Nevertheless, the initial density of the scenario varies as the agents move around the scene.

Regarding the movement pattern, we use wandering agents that move around the scene following random paths. This movement pattern generates the highest workload in the action server, since all the movements must be validated (Vigueras et al., 2008). Paths are generated by computing 100 random movements 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. We have compared the new GPU implementation with the baseline version, that is the reference implementation shown in the literature for collision tests (Par, 2008; Erra et al., 2009)

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

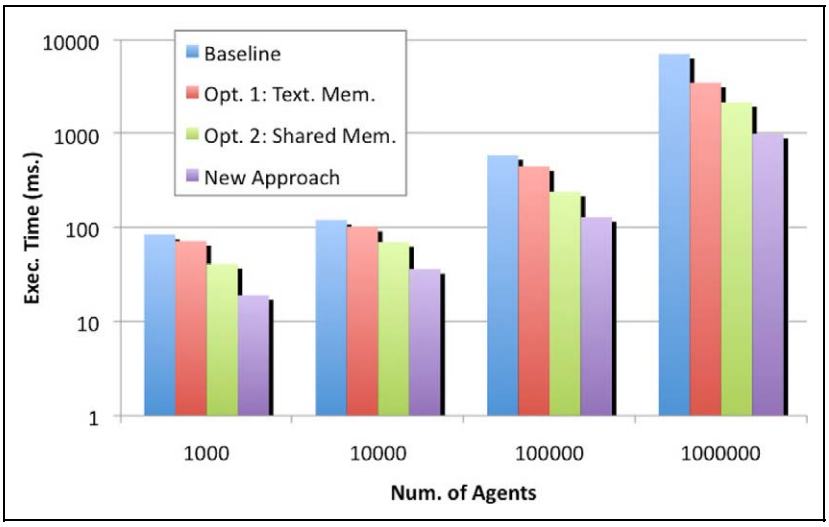

Figure 9. Execution times on the Tesla C870 card.

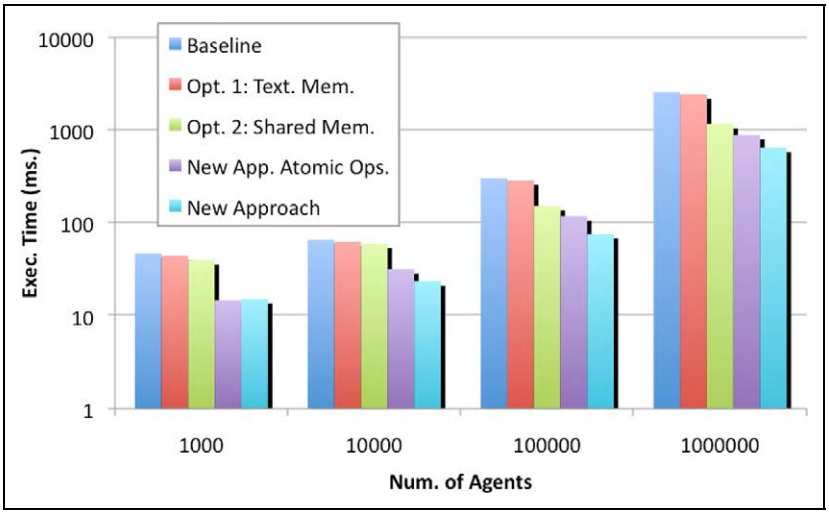

Figure 10. Execution times on the Tesla C1060 card.

Figures 9 and 10 compare the different GPU collision check implementations showing the overall execution time for each implementation. 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 on a log scale. Since the considered algorithm should scale up with the physical parallelism available on the GPU, we have considered two different NVIDIA Tesla GPUs: The Tesla C870 (16 SMs) and Tesla C1060 (30 SMs). When comparing the corresponding results for different cards, Figures 9 and 10 show that the execution times are inversely related to the number of SMs available on the cards.

Figure 9 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 operation. As would be expected, the greatest differences arise for the largest population size, that is, 1,000,000 agents. We use the texture memory to decrease the use of the device memory in the first optimization. In addition, we reduce the overhead due to kernel launches, resulting in a speedup of $2\times$ with respect to the baseline algorithm for the C870 card. In the second optimization, the shared memory is used in such a way that a coalesced access to device memory is guaranteed, obtaining a speedup of around $3.3\times$ compared to the baseline version. Nevertheless, the proposed technique achieves the best results, obtaining a speedup of $7.2\times$ with respect to the baseline version. Figure 10 shows that the effects of the texture memory optimization hardly arise for the case of the C1060 card. The reason is that for this card the global memory access algorithm allows us to obtain more coalesced accesses (NVIDIA, 2010), and therefore the baseline algorithm requires much shorter execution times than for the case of the C870 card. Nevertheless, the proposed algorithm achieves the best execution times, obtaining a speedup of $4.1\times$ for a population of 1,000,000 agents.

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

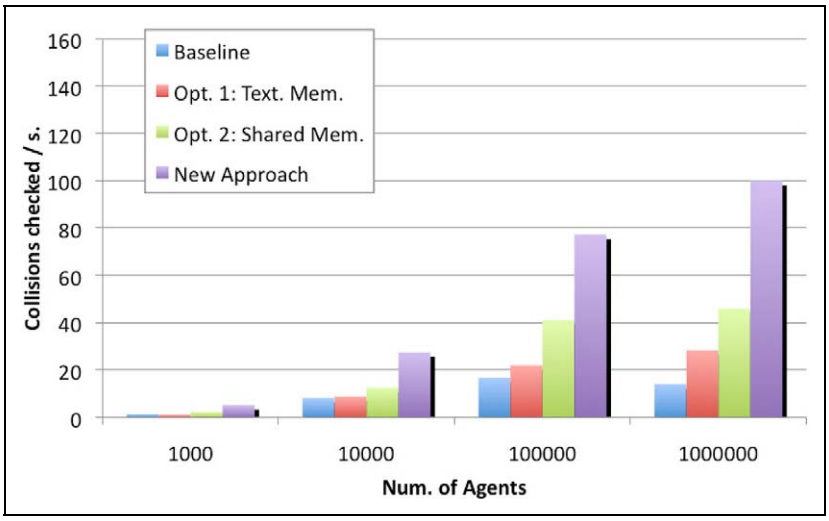

Figure 11. Collision rate on the Tesla C870 card.

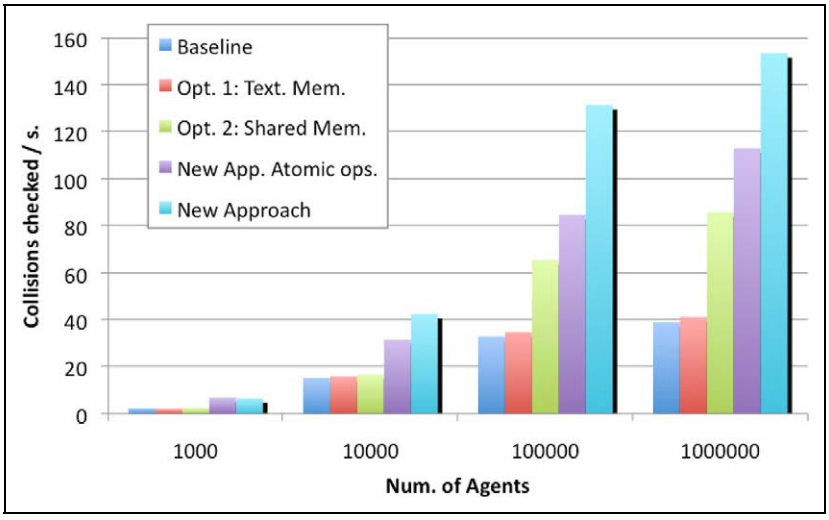

Figure 12. Collision rate on Tesla C1060 card.

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 the number of collisions checked per second. Figures 11 and 12 show the collisions check rates obtained when increasing the number of agents for both the Tesla C870 and C1060 cards, respectively. Figures 11 and 12 show that the proposed method without atomic operations performs at the highest collision check rate for all the population sizes. These figures also show that the collision check rate performed by the proposed method significantly increases with the number of available SMs on the GPU, assessing the scalability of this method.

The platform used for the CPU tests was a 16-core machine integrating 8 AMD Opteron processors (2 cores @ 1 GHz per processor), with 32.5 GB of RAM and the operating system was Linux 2.6.18-92. We used the POSIX API in order to obtain different configurations with an increasing number of cores. This API allows us to set the affinity of the execution threads, limiting the core set in which the threads are executed. Four configurations containing 2, 4, 8 and 16 cores were used in order to check the scalability with the number of cores of the CPU-based implementations.

Figure 13 shows the overall execution time for both the RCU- and the Mutex-based implementations. This figure shows on the -axis the number of agents considered for the simulations and the number of cores used for each population size (i.e. 2, 4, 8 and 16 cores). The *y*-axis shows, in log scale, the execution times measured in ms. It can be seen that the RCU obtains lower execution times when increasing the number of cores and as the population size grows. In addition, when the population size of the crowd grows, the differences in the execution time between the RCU and the Mutex implementations increases. The reason for this behavior is that the RCU synchronization method improves

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

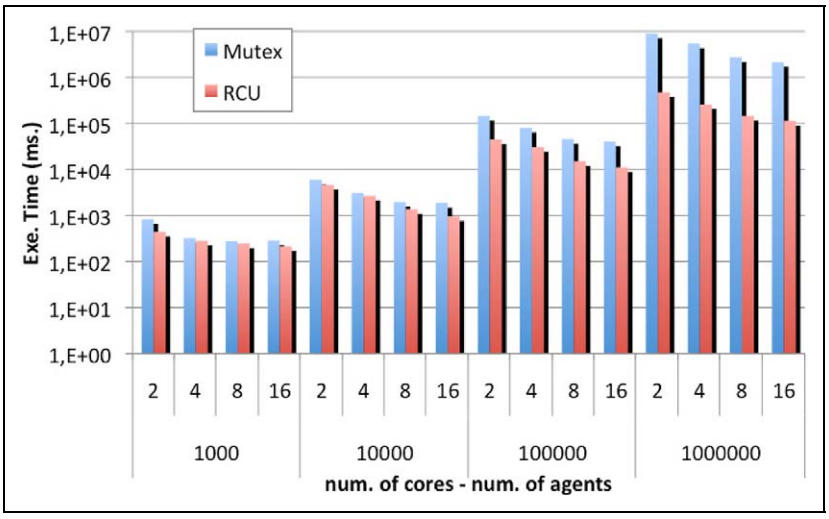

Figure 13. Overall execution time for both the RCU- and the Mutex-based implementations.

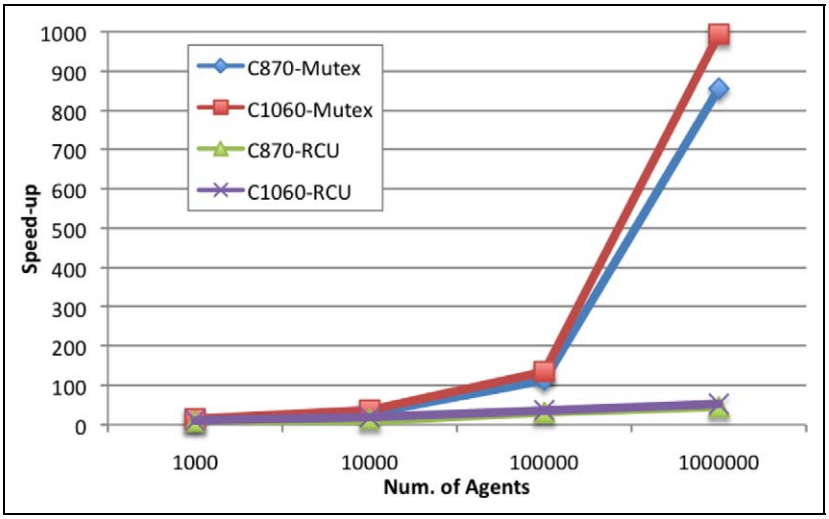

Figure 14. Speedup obtained for the new GPU procedure executed on different cards, with respect to the CPU implementations.

concurrency with respect to the Mutex method, since the critical section protected by a Mutex can only be sequentially executed by the existing threads.

Figure 14 shows the speedup obtained by the proposed GPU algorithm with respect to the CPU implementations. The results were obtained by comparing the execution times for both Tesla C870 and Tesla C1060 platforms with the execution times obtained for the CPU implementations using 16 cores. The execution times used for the GPU implementations were the aggregated execution time of the kernels, shown in Figures 9 and 10, plus the PCI time to transfer the data onto device memory and to return it back to the host. Figure 14 shows on the x-axis the number of agents considered for the simulations. The y-axis shows the speedup obtained. It can be seen that as the crowd population size increases, the speedup obtained for both GPU platforms in comparison with the RCU version, significantly decreases with respect to the speedup obtained for the Mutex version. In addition, the good scalability offered by the RCU version is the cause of the flatter slope of the speedup plot respect to the plot for the Mutex version. The reason for this behavior is that the RCU method allows read accesses in parallel with write accesses to dynamic data structures, avoiding the sequential access that a Mutex represents for these data structures. In this way, it can better exploit the existing number of processor cores, and the speedup of the GPU-based implementation hardly increases ($50\times$) with respect to the RCU-based implementation when the population size increases by orders of magnitude, obtaining a flat slope. In contrast, the Mutex implementation generates more serial accesses to data structures when the population increases, and therefore the performance improvement of the GPU implementations significantly increase with the population size with respect to the Mutex implementation. These results show the potential of the RCU synchronization method for improving parallel and distributed applications when executed on multi-core architectures. Nevertheless, despite the good scalability provided by the

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

Table 1. Theoretical specifications and speedups for the considered platforms.

	CPU Server	GPU C870	GPU C1060
Power (W)	1170	170	187.8
Theor. Peak perf (GFlops)	96	430	933
Perf. ratio	1	4.48	9.72
Speedup vs RCU	1	45.5	52.9
Speedup vs Mutex	1	850	1000

RCU implementation, the speedup obtained by the new GPU algorithm for the highest population size,with respect to the RCU implementation, is around 45.5× and 52.9× when compared to the Tesla C870 and C1060 platforms, respectively.

In order to measure the efficiency of the different platforms, Table 1 shows the power consumption and theoretical peak performance for the platforms considered, as well as the speedup achieved by the GPU implementations according to Figure 14. The first column shows the results for the multi-core server, consisting of 8 AMD Dual Opteron processors (16 cores), the second column shows the results for the NVIDIA C870 graphics card, and the last one shows the results for the NVIDIA C1060 graphics card.

Table 1 shows in the first row that the energy consumption of the multi-core server is one order of magnitude larger than the energy consumption of the graphics cards. However, the second row shows that the theoretical peak performance of the multi-core server is several times lower than that of the graphics cards. The third row expresses the relationships between the numbers in the second row as a ratio, showing that the theoretical peak performance of the C870 card is 4.48× higher than that of the multi-core server, and the theoretical peak performance of the C1060 card is 9.72× higher that of the multi-core server. Nevertheless, the fourth row shows that with a performance ratio of the hardware platform of 4.48 and 9.72×, the speedups obtained with respect to the RCU implementations are 45.5 and 52.9× for the C870 and C1060 cards, respectively (as shown in Figure 14). That is, the speedup achieved greatly exceeds the theoretical expected performance, multiplying it by a factor of around 5×. If we consider the speedup obtained by the graphics cards with respect to the performance of the multi-core server when using the Mutex implementation, then the achieved speedup multiplies the theoretical performance ratio by around 18×. These results validate the GPU as the best platform for performing collision check procedures in distributed crowd simulations.

6 Conclusions

In this paper, we have proposed different parallelization strategies for collision check procedures that take place in agent-based simulations, as part of a new distributed architecture for large-scale crowd simulations. These strategies are designed for exploiting the parallelism in both multi-core and many-core architectures like graphic processing units (GPUs). In order to fairly compare GPUs with multi-core implementations, we propose a parallel CPU version that uses RCU, a new synchronization method which significantly improves performance. In addition, we have presented a comparison study of the proposed implementations.

The comparison study shows that the GPU greatly accelerates the collision test with respect to any other implementation optimized for multi-core CPUs. On other hand, the comparison study shows the first performance evaluation of RCU in a real user-space application with complex data structures. The results show that RCU allows read accesses concurrently with write accesses to dynamic data structures, avoiding the sequential access that a Mutex represents for these data structures. In addition, we analyse the efficiency of the different platforms by comparing the performance provided by the different parallel implementations, taking into account the theoretical performance and power consumption of each platform. The performance evaluation results show that the GPU-based implementation consumes less energy and it provides a minimum speedup of 45× with respect to any of the CPU-based implementations.

Since interactivity is a hard constraint in crowd simulations, this reduction in the execution time of the collision check process represents a significant improvement of the overall system throughput and response time. Therefore, the simulations are significantly accelerated, and the system throughput and scalability are improved.

Funding

This work has been jointly supported by the Spanish MINECO and the European Commission FEDER funds (grant numbers CSD2006-00046 and TIN2009-14475-C04).

References

NVIDIA Corporation (2008). Particles Example. NVIDIA CUDA SDK.

Bleiweiss A (2008) GPU accelerated pathfinding. In: *23rd SIGGRAPH/EUROGRAPHICS symposium on graphics hardware (GH '08)*, Sarajevo, Bosnia, 20–21 June 2008, pp. 65–74. Aire-la-Ville: Eurographics Association.

Courty N and Musse SR (2005) Simulation of large crowds in emergency situations including gaseous phenomena. In: *computer graphics international (CGI '05)*, New York, USA, 22–24 June 2005, pp. 206–212. Piscataway: IEEE Press.

Desnoyers M (2009) Userspace RCU library: What linear multiprocessor scalability means for your application. In: *linux plumbers conference*, Portland, Oregon, USA, 23–25 September 2009.

Erra U, Frola B, Scarano V and Couzin I (2009) An efficient GPU implementation for large scale individual-based simulation of collective behavior. In: *international workshop on high performance computational systems biology (HiBi '09)*, Trento, Italy, 14–16th October 2009, pp. 51–58. USA, IEEE.

Gajinov V, Zyulkyarov F, Unsal OS, Cristal A, Ayguade E, Harris T, et al (2009) Quaketm: parallelizing a complex sequential application using transactional memory. In: *23rd international*

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

conference on supercomputing (ICS '09), New York, USA, 8–12 June 2009, pp. 126–135. New York: ACM Press.

Goswami P, Schlegel P, Solenthaler B and Pajarola R (2010) Interactive SPH simulation and rendering on the GPU. In: *SIGGRAPH/EUROGRAPHICS symposium on computer animation (SCA '10)*, Madrid, Spain, 2–4 July 2010, pp. 55–64. Aire-la-Ville: Eurographics Association.

Guy SJ, Chhugani J, Kim C, Satish N, Lin M, Manocha D, et al (2009) Clearpath: highly parallel collision avoidance for multi-agent simulation. In: *SIGGRAPH/EUROGRAPHICS symposium on computer animation (SCA '09)*, New Orleans, USA, 1–2 August 2009, pp. 177–187. New York: ACM Press.

Harada T, Tanaka M, Koshizuka S and Kawaguchi Y (2007) Real-time rigid body simulation using GPUs. *IPSJ SIG Technical Reports* 13: 79–84.

Hart TE, McKenney PE and Brown AD (2006) Making lockless synchronization fast: Performance implications of memory reclamation. In: *20th IEEE international parallel and distributed processing symposium*, Rhodes, Greece, 25–29 April 2006, Piscataway: IEEE Press.

Helbing D, Farkas I and Vicsek T (2000) Simulating dynamical features of escape panic. *Nature* 407: 487–490.

Herault A, Bilotta G and Dalrymple RA (2010) SPH on GPU with CUDA. *Journal of Hydraulic Research* 48: 74–79.

Kim D, Heo JP, Huh J, Kim J and Yoon SE (2009) HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. *Computer Graphics Forum* 28(7): 1791–1800.

Latta L (2004) Building a million particle system. In: *game developers conference (GDC '04)*, San Jose, California, USA, 22–26 March 2004.

Lauterbach C, Garland M, Sengupta S, Luebke D and Manocha D (2009) Fast BVH construction on GPUs. *Computer Graphics Forum* 28(2): 375–384.

Lauterbach C, Mo Q and Manocha D (2010) Gproximity: Hierarchical GPU-based operations for collision and distance queries. *Computer Graphics Forum* 29(2): 419–428.

Lozano M, Morillo P, Orduña JM, Cavero V and Vigueras G (2009) A new system architecture for crowd simulation. *Journal of Network and Computer Applications* 32(2): 474–482.

Lysenko M and D'Souza RM (2008) A framework for mega-scale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation* 11(4): 10.

McKenney PE and Slingwine JD (1998) Read-copy update: Using execution history to solve concurrency problems. In: *18th international conference on parallel and distributed computing systems (PDCS '98)*, Las Vegas, USA, 12–14 September 2005, pp. 509–518. IASTED Acta Press.

[NVIDIA(2010)] NVIDIA (2010) CUDA programming guide 3.2. Available at: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (accessed 1 February 2012).

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

[Perumalla and Aaby(2008)] Perumalla KS and Aaby BG (2008) Data parallel execution challenges and runtime performance of agent simulations on GPUs. In: *2008 spring simulation multiconference (SpringSim '08)*, Ottawa, Canada, 14–17 April 2008, pp. 116–123. New York: ACM Press.

[Peter et ~ al.(2004)Peter, Mark, and Rudiger] Peter K, Mark S and Rudiger W (2004) Uberflow: a GPU-based particle engine. In: *19th SIGGRAPH/EUROGRAPHICS symposium on graphics hardware (GH '04)*, Grenoble, France, 29–30 August 2004, pp. 115–112. New York: ACM Press.

[Pratas et ~ al.(2009)Pratas, Trancoso, Stamatakis, and Sousa] Pratas F, Trancoso P, Stamatakis A and Sousa L (2009) Fine-grain parallelism using multi-core, Cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In: *38th international conference on parallel processing (ICPP '09)*, Vienna, Austria, 22–25 September 2009, pp. 9–17. Piscataway: IEEE Press.

[Reynolds(2006)] Reynolds C (2006) Big fast crowds on PS3. In: *1st ACM/SIGGRAPH symposium on videogames*, Boston, MA, USA, 29–30 July 2006, pp. 113–121. New York: ACM Press.

[Reynolds(1987)] Reynolds CW (1987) Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics* 21(4): 25–34.

[Satish et ~ al.(2009)Satish, Harris, and Garland] Satish N, Harris M and Garland M (2009) Designing efficient sorting algorithms for manycore GPUs. In: *2009 IEEE international symposium on parallel and distributed processing (IPDPS '09)*, Rome, Italy, 23–29 May 2009, pp. 1–10. Piscataway: IEEE Press.

[Sundell and Tsigas(2008)] Sundell H and Tsigas P (2008) Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing* 68(7): 1008–1020.

[Teschner et ~ al.(2003)Teschner, Heidelberger, Mueller, Pomeranets, and Gross] Teschner M, Heidelberger B, Mueller M, Pomeranets D and Gross M (2003) Optimized spatial hashing for collision detection of deformable objects. In: *vision, modeling, and visualization conference (VMV '03)*, München, Germany, AKA Gmbh, 19–21 November 2003, pp. 47–54. Eurographics Association.

[van ~ den Berg et ~ al.(2008)van ~ den Berg, Lin, and Manocha] van den Berg J, Lin MC and Manocha D (2008) Reciprocal velocity obstacles for real-time multi-agent navigation. In: *IEEE international conference on robotics and automation (ICRA '08)*, Pasadena, California, USA, 19–23 May 2008, pp. 1928–1935. Piscataway: IEEE Press.

[Vigueras et ~ al.(2010)Vigueras, Lozano, Ordu na, and Grimaldo] Vigueras G, Lozano M, Orduña JM and Grimaldo F (2010 a) A comparative study of partitioning methods for crowd simulations. *Journal of Applied Soft Computing* 10(1): 225–235.

[Vigueras et ~ al.(2008)Vigueras, Lozano, Perez, and Ordu na] Vigueras G, Lozano M, Perez C and Orduña J (2008) A scalable architecture for crowd simulation: Implementing a parallel action server. In: *37th international conference on parallel processing (ICPP '08)*, Portlan, Oregon, USA, 8–12 September 2008,pp. 430–437. Los Alamitos, USA: IEEE Press.

[Vigueras et ~ al.(2010)Vigueras, Ordu na, and Lozano] Vigueras G, Orduña J and Lozano M (2010 b) A GPU-based multi-

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

agent system for real-time simulations. In: *8th international conference on practical applications of agents and multiagent systems (PAAMS '10)*, Salamanca, Spain, 26–28 April 2010, vol. 70, pp. 15–24. New York: Springer.

[Vigueras et∼al.(2011)Vigueras, Ordu na, Lozano, and Chrysanthou] Vigueras G, Orduña JM, Lozano M and Chrysanthou Y (2011) A distributed visualization system for crowd simulation. *Integrated Computer-Aided Engineering* 18(4): 1008–1020.

[Vigueras et∼al.(2010)Vigueras, Ordu na, Lozano, Cecilia, and Garc] Vigueras G, Orduña JM, Lozano M, Cecilia JM and García JM (2010 c) Improving the GPU-based collision check procedure for distributed crowd simulations. In: *19th international conference on parallel architectures and compilations techniques (PACT '10)*, Vienna, Austria, 11–15 September 2010.

[Xu et∼al. (2009) Xu, R.∼Kirk, and Jenkins] Xu CR, Kirk S and Jenkins S (2009) Tiling for performance tuning on different models of GPUs. In: *2nd international symposium on information science and engineering (ISISE '09)*, Shanghai, China, 26–28 December 2009, pp. 500–504. Washington, USA: IEEE Computer Society.

[Zhou et∼al. (2008) Zhou, Hou, Wang, and Guo] Zhou K, Hou Q, Wang R and Guo B (2008) Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics* 27(5): 1–11.

[Zyulkyarov et∼al. (2009) Zyulkyarov, Gajinov, Unsal, Cristal, Ayguadà Harris, and Valero] Zyulkyarov F, Gajinov V, Unsal OS, Cristal A, Ayguadé E, Harris T, et al. (2009) Atomic quake: using transactional memory in an interactive multiplayer game server. In: *14th ACM/SIGPLAN symposium on principles and practice of parallel programming (PPoPP '09)*, Raleigh, North Carolina, USA, 14–18 February 2009, pp. 25–34. New York: ACM Press.

Author biographies

Guillermo Vigueras received an MSc degree in computer engineering from the University of Valencia, Spain in 2005. He is a member of the GREV research group which is part of the ACCA team. Currently, he is a PhD candidate at the University of Valencia, Spain. His research interests include crowd simulations, parallel programming and distributed systems.

Juan Orduña received an MSc degree in computer engineering from the Technical University of Valencia, Spain in 1990 and a PhD in computer engineering from the University of Valencia, Spain in 1998. His research has been developed inside the ACCA team. He was a Computer Engineer with Telefónica de España, Manpel Electrónica, and at the Technical University of Valencia. He is currently a Lecturer Professor with the Department of Informatics, University of Valencia, where he leads the Networking and Virtual Environments Group. He is a member of the HiPEAC network of excellence, and his research is currently supported by the Spanish MEC and the European Commission through several projects. His research currently focuses on networks-on-chip, distributed virtual environments, and crowd simulations. He has published papers about his research in a number of international journals and conferences. He has served as a Program Committee Member for different conferences and workshops (e.g. the IEEE International Conference on Parallel Processing, European Conference on Parallel Processing, IEEE Virtual Reality Conference, and International Conference on Parallel and Distributed Systems), and is also a reviewer for journals such as *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Network and Computer Applications* and the *IEEE Journal of Selected Areas in Communication*.

Miguel Lozano received an MSc degree in computer engineering from the Technical University of Valencia, Spain in 1996. Since 2006 he has been an Associate Professor of Computer Science and Artificial Intelligence. He received a PhD in computer engineering from the University of Valencia in 2005 and his current research interests include large-scale multi-agent systems, social engineering and distributed/parallel architectures. He is an active member of the Networks and Virtual Environments Group and has published papers on intelligent multi-agent decision taking and large-scale distributed simulations in a large number of high-impact international journals and conferences.

José Cecilia received a BSc degree in computer science from the University of Murcia, Spain in 2005 and an MSc degree in computer science from Cranfield University, UK in 2007. He is currently a PhD candidate at the University of Murcia. He has recently received a collaboration grant from the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) to visit the Novel Computation Group at Manchester Metropolitan University, UK. He has published several papers in international peer-reviewed journals and conferences. His research interests include heterogeneous architectures as well as bio-inspired algorithms for evaluating the newest frontiers of computing.

Jose García received an MSc degree in electrical engineering and a PhD in computer engineering both from the Technical University of Valencia, Spain in 1987 and 1991, respectively. He is Professor of Computer Architecture at the Department of Computer Engineering, and also Head of the Research Group on Parallel Computer Architecture. He is currently serving as Dean of the School of Computer Science at the University of Murcia, Spain. He has developed several courses on Computer Structure,

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014

Peripheral Devices, Computer Architecture, Parallel Computer Architecture and Multicomputer Design. He specializes in computer architecture, parallel processing and interconnection networks. His current research interests lie in high-performance coherence protocols for chip multiprocessors (CMPs) and shared-memory multiprocessor systems, high-speed interconnection networks, and the use of GPUs for general-purpose applications. He has published more than 110 refereed papers in different journals and conferences in these fields. He is member of HiPEAC and also member of several international associations such as the IEEE and ACM.

Downloaded from hpc.sagepub.com at Universidad de Murcia on January 23, 2014