A GPU-Based Multi-Agent System for Real-Time Simulations

Guillermo Vigueras and Juan M. Orduña and Miguel Lozano

Abstract The huge number of cores existing in current Graphics Processor Units (GPUs) provides these devices with computing capabilities that can be exploited by distributed applications. In particular, these capabilites have been used in crowd simulations for enhancing the crowd rendering, and even for simulating continuum crowds. However, GPUs have not been used for simulating large crowds of complex agents, since these simulations require distributed architectures that can support huge amounts of agents. In this paper, we propose a GPU-based multi-agent system for crowd simulation. Concretely, we propose the use of an on-board GPU to implement some of the tasks that a distributed server for crowd simulations should perform. The huge number of cores in the GPU is used to simultaneously validate movement requests from different agents, greatly reducing the server response time. Since this task represents the critical data path, the use of this hardware significantly increases the parallelism achieved with respect to the implementation of the same distributed server on a CPU. An application example shows that the system can support agents with complex navigational behaviors.

1 Introduction

The huge number of cores existing in current Graphics Processor Units (GPUs) provides these devices with computing capabilities that can be exploited by distributed applications. Since some years ago, GPU vendors introduced programmability to these devices in order to facilitate their use for scientific computation. One of the

Guillermo Vigueras and Juan M. Orduña and Miguel Lozano
Departamento de Informática, Universidad de Valencia
Avda. Vicente Andrés Estellés, s/n
Burjassot (Valencia - SPAIN
e-mail: {guillermo.vigueras, juan.orduna,miguel.lozano}@uv.es

distributed applications that can benefit from the capabilities of the GPUs is crowd simulation.

Regarding crowd simulations, some proposals have been made for exploiting the capabilities of multicore architectures. In this sense, a new approach has been presented for PLAYSTATION3 to distribute the load among the PS3-Cell elements[13]. Another work uses graphics hardware to simulate crowds of thousands individuals using models designed for gaseous phenomena [1]. Recently, some authors have started to use GPU in an animation context (particle engine) [11, 5], and there are also some proposals for running simple stochastic agent simulations on GPUs [7, 10]. However, these proposals are far from displaying complex behaviors at interactive rates. On the other hand, other proposals including complex agent systems have been made [9, 15], but they are not designed to provide the required scalability in number of agents.

In this paper, we show that large scale multiagent systems can benefit from the use of GPU computing. In order to achieve this goal, we have implemented a distributed server for crowd simulations [16] using an on-board GPU. The huge number of cores in the GPU is used to simultaneously validate movement requests from different agents, greatly reducing the server response time. Since this task represents the critical data path, the use of this hardware significantly increases the parallelism achieved with respect to the implementation of the same multiagent system using a distributed server implemented on a CPU. Thus, the performance evaluation results show that the system throughput is significantly increased, supporting a significantly higher number of agents while providing the same latency levels. These results can be used for simulating crowds with a larger size.

The rest of the paper is organized as follows: Section 2 describes in detail the use of GPUs for increasing parallelism in crowd simulations. Next, Section 3 shows the performance evaluation of the proposed architecture. Next, Section 4 describes an application example to show our system working in a real and complex scenario. Finally, Section 5 shows some concluding remarks and future work to be done.

2 A GPU-based Action Server for Crowd Simulation

In a previous work, a distributed system architecture for crowd simulation was proposed in order to take advantage of the underlying distributed computer system [16]. That software architecture is mainly composed by two elements: the action server (AS) and the client processes (CP). The AS is devoted to execute the crowd actions, while a CP handles a subset of the existing agents. Agents are implemented as threads of a single process for reducing the communication cost. Each thread manages the perception of the environment and the reasoning about the next action. Each client process is hosted on a different computer, in such a way that the system can have a different number of client processes, depending on the number of agents in the system. The Action Server is divided into a set of processes so that each one can be executed in parallel in a different computer. Each of these processes is denoted as an Action Server, while the whole set of ASs is denoted as the Parallel Action Server (PAS). The rest of the section describes the version of the PAS for execution in CPU, and the modifications made to this version in order to take advantage of a GPU.

Each piece (process) of the Parallel Action Server can be viewed as a partial world manager, since it controls and properly modifies the information in a region of the whole simulation space. Thus, it can be considered as the system core. Each AS process contains three basic elements: the Interface module, the Crowd AS Control (CASC) module and the Semantic Data Base (SDB). Figure 1 illustrates a detailed scheme of an AS.

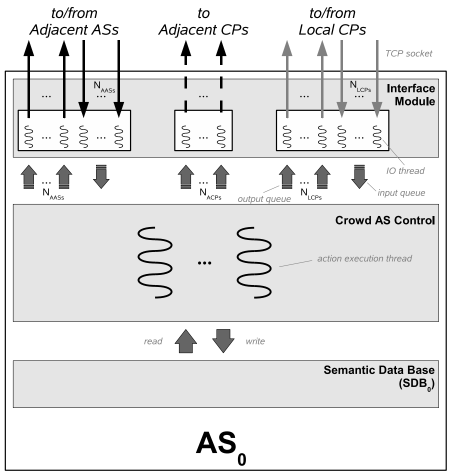

Fig. 1 Internal structure of an Action Server

The main module is the Crowd AS Control module, which is responsible for executing the crowd actions. This module contains a configurable number of threads for executing actions (action execution threads in Figure 1). For an action execution thread (AE thread), all messages sent to or received from other ASs and CPs are exchanged asynchronously (the details are hidden by the Interface module, see below). This means that the AE threads only may have to wait when accessing shared data structures such as the semantic database. Thus, experimental tests have shown that having more AE threads than cores allows each AS to take advantage of several cores.

Most action requests from agents are executed from start to end by the AE thread, that extracts the requests from the corresponding input queue and process them. These requests consist of collision tests, in order to check if the new position computed by each agent when it moves coincides with the position of other agent or object. The Interface module hides all the details of the message exchanges. This module provides the Crowd AS Control module with the abstraction of asynchronous messages. Two separate input queues exist, one for messages coming from local CPs (action requests) and the other one for messages coming from adjacent ASs (responses to requests issued because of local border actions or requests for remote border actions from adjacent ASs). Having two separate input queues is an efficient way of giving a higher priority to messages from adjacent ASs. The reason for improving the priority of these messages is that the border actions are the ones whose processing takes longer, and we should reduce as much as possible their response time to provide realistic interactive effects.

In order to process messages as soon as they arrive, the Interface module contains one IO thread dedicated to getting incoming messages from each TCP socket. There are no input threads associated with sockets connecting one AS to their adjacent CPs, because CPs only send messages to their local AS. In the same way, there is one IO thread and one output queue per TCP socket, so that messages are sent as soon as the corresponding TCP socket is ready for writing.

The GPU used for implementing a parallel action server has been a NVIDIA Tesla C870. This GPU has 128 thread processors, each one with its texture cache and low-latency shared memory. The communications of the different shared memories with global memory are performed at the same time, by means of Load/Store operations. Each thread processor executes a kernel in parallel with the rest of processors. Just before calling this kernel, the use of *cudaBindTexture()* instruction allows to copy into each texture cache the proper range of global memory addresses. In this way, each thread processor can access to local data with a very low latency. The integration of a huge number of cores and on-chip memories with low latency in this GPU allows to perform up to 128 collision tests in parallel, clearly outperforming the number of collision tests performed in parallel when using the CPU. Additionally, the SIMD structure of the GPU avoids mutual exclusions when accessing shared data structures, while the CPU-version AE Threads have to wait when accessing data structures shared among the existing cores, such as the semantic database.

A general overview of the collision checking process in the server is as following: agent requests received by the interface module are passed to the CASC module. However, the AE threads in the GPU-based server do not actually perform the collision tests (unlike the AE Threads in the CPU-based server). Instead, the AE threads collect the requests from the interface module and copy them to the SDB module as they arrive, until $N_{OPERATIONS}$ requests are collected. At this point, one of the AE threads signals the *GPU manager thread*. This thread controls the data path in the GPU. Concretely, this thread copies the requests into the GPU global memory and launches the collision tests. When the collision tests finish, the GPU manager thread copies the results into the CPU memory. When the AE threads finish their current task, they will start to process the GPU replies. That is, the AE threads in the GPU-based server collect client requests, but these requests are actually checked by the GPU manager thread. In this way, the parallelism achieved by having several AE Threads is decoupled from the parallelism provided by the GPU thread processors. This collision checking scheme can take advantage of the parallel computation capabilities of a GPU like the NVIDIA Tesla C870.

The internal structure of a parallel action server using a GPU is shown in Figure 2. In this implementation, the SDB module contains the *object positions array*, the *collision response array* and the *GPU manager thread*. The *object positions array* is the host-GPU input interface. This array, contains the agents positions, needed to check collisions. The structure *collision response array*, represents the host-GPU output interface, and it contains the result of the collisions tests performed by the GPU. When the *GPU manager thread* is signaled by the AE Threads, it copies the object positions array into the GPU and launches the collision tests. When the collision tests finish, the *GPU manager thread* copies the results from the GPU memory into the SDB module. Then, it signals the AE threads to start replying to CPs or ASs.

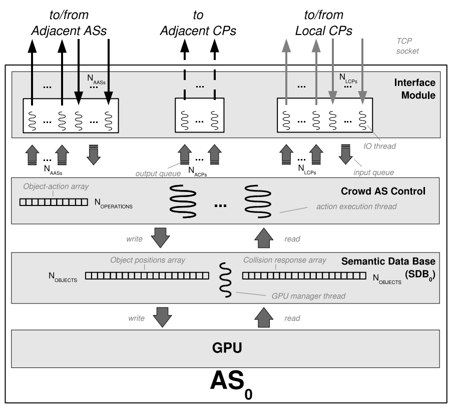

Fig. 2 Internal structure of an action server using a GPU

The *object positions array* has $N_{OBJECTS}$ elements, that is, as many elements as agents managed by a server. An agent identifier is associated to each request, in order to update the proper array position. Each element in the array has four floats for each object passed to the GPU. The first and second floats contain the x and y coordinates of the agent's position, respectively. The third float is not used. The fourth float is a flag indicating the GPU whether the element in the array has been updated by AE threads. If an object position has been updated (i.e. the flag is equal to a positive number representing the agent identifier number) then the collision test will be performed. Otherwise (i.e. the flag is equal to -1.0) the collision test is skipped. This flag is initialized by AE threads when the x and y coordinates are updated and is cleared when a collision test finishes.

The *collision response array* has, as the previous array, $N_{OBJECTS}$ elements. Each element of the array has two floats. The first one indicates whether a collision occurred (in this case is equals to 1.0) or not (in this case is equals to 0.0). The second float contains the agent identification number and indicates which agent is associated to the collision result. The agent identification number is obtained by the GPU from the fourth float contained in each element of the *object positions array*. The *collision response array* is accessed by the AE threads to send the collision responses corresponding to each collision request. Although this array has as many elements as agents, the AE threads will only collect $N_{OPERATIONS}$ instead of

$N_{OBJECTS}$ collision results. In order to efficiently read the *collision response array*, random access is needed. The *object-action array* (that provides this random access) contains the elements that should be accessed for reading each collision result.

Collisions on the GPU are checked using a spatial hashing based method. For this reason, a two dimensional grid is created in the GPU memory when the simulation starts. The dimensions of the grid, grid cell size and grid origin coordinates are fixed, depending on the scene simulated. The easiest way to implement the collision grid into the GPU is defining an array in which each position represents a grid cell. The mapping of agents to grid cells is performed by the spatial hashing method, depending on the cell size and the position of agents. Because many agents can fall within the same cell, the GPU threads can simultaneously update the same array position. Atomic operations are needed in the GPU in order to allow simultaneous accesses to global memory [8]. However, the GPU used in our implementation does not support atomic operations. For that reason, a more complex approach (based on sorting) has been implemented, using a fast radix sort method [3]. In this approach, there is a global memory array (denoted as *ObjectPositionsArray*) containing the agents positions. Another array (denoted as *collisionResponse*) contains the collision results. Also, some other structures are used: *ObjectsHash*, *sortedPositions* and *cellStart*. The *ObjectsHash* array represents the collision grid, and it stores the cell to which each agent belongs. Concretely, it contains a pair *(cell identifier, agent identifier)* for each position. The *sortedPositions* array contains the same elements as *ObjectPositionsArray*, but sorted by cell identifier. In this way, neighboring agents can be efficiently obtained for collision checking. The *cellStart* array allows to determine the beginning of each cell in the *ObjectsHash* array. Thus, if position i in *cellStart* array contains the value j, it means that the first agent of grid cell i appears in position j in the *ObjectsHash* array. In this way, the *cellStart* array allows a quick access to the agents in neighboring cells.

3 Performance Evaluation

This section shows the performance evaluation of the GPU-based server described in the previous section. We have performed different measurements on a real system using the GPU-based server. For comparison purposes, we have also performed the same measurements on the same real system but using the CPU-based server. The most important performance measurements in distributed systems are latency and throughput [2]. The performance improvement that a GPU-based server can provide to the distributed crowd simulation [16] would depend on the number of distributed servers in the system. In order to evaluate the worst case, we have performed simulations with one server and with different number of agents. Concretely, we have measured the aggregated computing time for collision tests during the simulations, and the average response times provided to agents. In order to define an acceptable behavior for the system, we have considered 250 ms. as the maximum threshold value for the average response time. This value is considered as the limit for providing realistic effects to users in Distributed Virtual Environments (DVEs) [4, 14]. Since crowd simulation can be considered as DVEs where avatars are intelligent agents (instead of dummy entities controlled by users), we have used this value taken from the literature.

We have performed crowd simulations with wandering agents because all their actions should be verified by an AS. Each simulation consists of the crowd moving within the virtual world following *k*-length random paths. Nevertheless, in order to obtain reproducible and comparable results, we have saved the paths followed by the agents in the first execution of each configuration and we have used the same paths for all the executions tested with the same number of avatars. For all the populations tested, the average response time has become stable within the first thirty seconds of execution time. Therefore, we have used simulations lengths of thirty seconds for the configurations tested.

We have performed experiments using a computer platform with one server and six clients. The server was based on Intel Core Duo 2.0 GHz, with 4GB of RAM, executing Linux 2.6.18.2-34 operating system and had incorporated a NVIDIA C870 Tesla GPU. Each client computer was based on AMD Opteron (2 x 1.56 GHz processors) with 3.84GB of RAM, executing Linux 2.6.18-92 operating system. The interconnection network was a Gigabit Ethernet network. Using this platform, we have simulated up to nine thousand agents.

Figure 3 a) shows the aggregated computing time for operations involved in collision tests during the whole simulation. On the X-axis, this figure shows the number of agents in the system for different simulations. The Y-axis shows aggregated computing time (in seconds) devoted to compute the collision tests required by the simulations. This Figure shows that the plot for the CPU-based server has a parabolic shape, while the plot for the GPU-based server has a flat slope. These results show that the use of the replicated hardware in the GPU has a significant effect in the time required by the server to compute the collision tests. The population size (number of agents) considered in these simulations generates a number of collision tests that does not exceed the computation bandwidth available in the GPU. As a result, the computing time required for different population sizes is very similar. An additional benefit is derived from the fact that the GPU is exclusively devoted to compute collision tests, releasing the CPU from that task. This is the reason for the lower values shown by the GPU plot, even for the smallest population sizes.

Although Figure 3 a) shows a huge improvement in the GPU-based server performance, the effects of such improvement should be measured on the system. Thus, Figure 3 b) shows the average response time provided for the agents hosted by a given client. In order to show the results for the worst case, we show the values for the client with the highest average response time (it must be noticed that we have a single server and six clients).

Figure 3 b) shows that the average response times increases linearly with the number of agents for both plots until 6000 agents. From that point up, the same behavior is shown when no GPU is used. However, when using GPU the plot shows a flat slope. Beyond a given threshold, the average response time is not increased because all the collisions tests are performed in parallel in the GPU. Additionally,

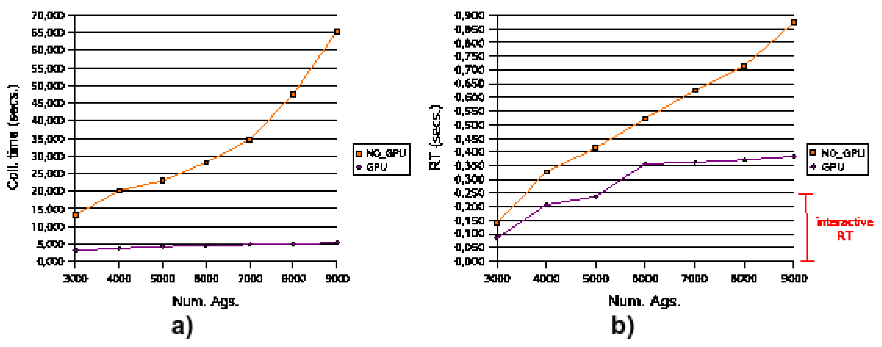

Fig. 3 a) Aggregated computing time for collision tests. b) Average Response times provided to agents.

Figure 3 b) shows that the CPU-based crowd simulation can support up to 3500 agents while providing average response times below the 250 milliseconds threshold. When GPU is used, the number of agents supported grows up to 5300 agents, providing an improvement of a 50%. Taking into account that this improvement can be achieved per each server in the system, these results show that the GPU-based server can actually have a significant impact in the performance of large-scale crowd simulations.

Additionally, it must be noticed that the performance achieved when using GPUs depends on the number of cores in the CPU, because the CPU threads are responsible for replying to the agents requests. Since we have used dual-core processors for evaluation purposes (in order to measure the worst-case performance), the improvements shown in this section can be increased when using platforms with a higher number of processor cores.

4 Application Example

In Section 3 we have shown the performance evaluation of our proposal using wandering agents, since this kind of agent generates the highest load in the server because all its actions should be verified by an AS. In this section, we show that the proposed approach can integrate more complex navigational behaviors in real and structured scenarios. Concretely, we have simulated the evacuation of an actually existing Faculty building.

We have captured different simulation data in order to visualize the movement followed by the crowd. These data are computed in the Client Processes, in such a way that the Action Server is not affected by these computations. The behavior of agents has been integrated in the client processes, as explained in Section 2. A hybrid navigation, composed of a two-modules model, has been used. On one hand, the high-level navigation module is in charge of pre-computing a set of paths from any cell to the exits. We have implemented this module as a Cellular Automata (CA). The number and length of the paths computed can be adjusted in order to reduce the memory used by the CA [6]. On other hand, a low-level navigation module determines how the paths computed by the CA should be followed. This module is implemented as a rule-based model [9, 12], and it allows to move agents in a continuum domain.

In this application example there are also static objects that agents should avoid. Since the rule-based navigation model implemented in the client process can provide agent positions colliding with static objects, some modifications are needed in the GPU collision checking algorithm. However, the performance of the GPU collision checking algorithm is not significantly affected by these changes, since the obstacles grid is computed and sorted before the simulation starts, and the grid data is kept in GPU memory during the whole simulation, avoiding memory transfers between CPU and GPU. Figure 4 shows a detailed 3D view of a congestion produced during the evacuation simulation. This example shows that the performance improvements shown in Section 3 can also be obtained when simulating agents with complex navigational behaviors. Moreover, these improvements are achieved without affecting the visual quality of the crowd.

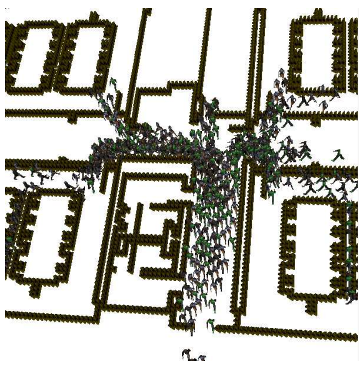
Fig. 4 3D snapshot of the evacuation scenario.

5 Conclusions and Future Work

In this paper, we have proposed the implementation of a distributed server for crowd simulations using an on-board GPU. The huge number of cores in the GPU is used to simultaneously validate movement requests from different agents, greatly reducing the server response time. Since this task represents the critical data path, the use of this hardware significantly increases the parallelism achieved with respect to the implementation of the same distributed server on a CPU. Thus, the system can support a significantly higher number of agents while providing the same latency levels.

As a future work to be done, we plan to study the performance improvements that the use of processors with a higher number of cores can provide.

Acknowledgements This work has been jointly supported by the Spanish MEC and the European Commission FEDER funds under grants Consolider-Ingenio 2010 CSD2006-00046 and TIN2009-14475-C04-04.

References

1. Courty, N., Musse, S.R.: Simulation of large crowds in emergency situations including gaseous phenomena. In: CGI '05: Proceedings of the Computer Graphics International 2005, pp. 206–212. IEEE Computer Society (2005)
2. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks: An Engineering Approach. IEEE Computer Society Press (1997)
3. Harada, T., Tanaka, M., Koshizuka, S., Kawaguchi, Y.: Real-time rigid body simulation using gpus. IPSJ SIG Technical Reports **13**, 79–84 (2007)
4. Henderson, T., Bhatti, S.: Networked games: a qos-sensitive application for qos-insensitive users? In: Proceedings of the ACM SIGCOMM 2003, pp. 141–147 (2003)
5. Latta, L.: Building a million particle system. In: In Proc. of Game Developers Conference(GDC-04) (2004)
6. Lozano, M., Morillo, P., Orduña, J.M., Cavero, V., Vigueras, G.: A new system architecture for crowd simulation. J. Netw. Comput. Appl. **32**(2), 474–482 (2009). DOI http://dx.doi.org/10.1016/j.jnca.2008.02.011
7. Lysenko, M., D'Souza, R.M.: A framework for megascale agent based model simulations on graphics processing units. Journal of Artificial Societies and Social Simulation **11**(4), 10 (2008). URL http://jasss.soc.surrey.ac.uk/11/4/10.html
8. NVIDIA Corporation: NVIDIA CUDA Programming Guide (2007). Ver. 1.1
9. Pelechano, N., Allbeck, J.M., Badler, N.I.: Controlling individual agents in high-density crowd simulation. In: SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, pp. 99–108. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2007)
10. Perumalla, K.S., Aaby, B.G.: Data parallel execution challenges and runtime performance of agent simulations on gpus. In: SpringSim '08: Proc. of Spring simulation multiconference, pp. 116–123. ACM, New York, NY, USA (2008)
11. Peter, K., Mark, S., Rudiger, W.: Uberflow: a gpu-based particle engine. In: HWWS '04: Proc. of the SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. ACM (2004)
12. Reynolds, C.: Steering behaviors for autonomous characters. In: Game Developers Conference 1999, pp. 763–782 (1999)
13. Reynolds, C.: Big fast crowds on ps3. In: Proceedings of the ACM SIGGRAPH symposium on Videogames, pp. 113–121. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1183316.1183333
14. Rueda, S., Morillo, P., Orduña, J.M.: A comparative study of awareness methods for peer-to-peer distributed virtual environments. Comput. Animat. Virtual Worlds **19**(5), 537–552 (2008). DOI http://dx.doi.org/10.1002/cav.v19:5
15. Treuille, A., Cooper, S., Popovic, Z.: Continuum crowds. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, pp. 1160–1168. ACM (2006)
16. Vigueras, G., Lozano, M., Perez, C., Orduña, J.: A scalable architecture for crowd simulation: Implementing a parallel action server. In: Proceedings of the 37th International Conference on Parallel Processing (ICPP-08), pp. 430–437 (2008). DOI 10.1109/ICPP.2008.20