A Novel Data Structure for Particle System Simulation based on GPU with the Use of Neighborhood Grids

Mark Joselli
Jose Ricardo Silva Junior
Marcelo Zamith
Esteban Clua
MediaLab, IC-UFF

Eduardo Soluri
Nullpointer Tecnologia

Abstract

Simulation and visualization of particles in real-time can be a computationally intensive task. This intensity comes from diverse factories, being one of them is the $O(n^2)$ complexity of the traversal algorithm, necessary for the proximity queries of all pair of particles that decide the need to compute collisions. Previous works reduced this complexity by considerably factors, using adequate data structures for spatial subdivision and parallel computing on modern graphic hardware, achieving interactive frame rates in real-time simulations. However, the performance of existent proposals are heavily affected by the maximum density of the spatial subdivision cells, which is usually high, yet leading to algorithms that are not optimal. In this paper we apply a novel data structure, which is called neighborhood grid, and a simulation architecture that provides for extremely low parallel complexity. Also, we compared this work with the traditional spatial hashing achieving a speedup up to 9.5 with a similar visual experience and with lesser use of memory.

Keywords:: Particle Simulation, Real-Time Simulation, GPGPU, Data Structure, Cellular Automata.

Author's Contact:

{mjoselli, jricardo, mzamith, esteban}@ic.uff.br
esoluri@nullpointer.com.br

1 Introduction

The increase of the level of realism in virtual simulation depends not only on the enhancement of modeling and rendering effects, but also on the improvement of different aspects such as animation, artificial intelligence of the characters and physics simulation.

GPUs are a collection of SIMD *(Single Instruction Multiple Data)* processors designed to run streamed graphics pipelines, a computation model where the processing of each pixel is independent of the others and usually requires localized memory reads (texture fetching). There are rules of thumb to create efficient streamed applications, where, the most important one is to organize the data streams in a way that maximizes memory read performance based on locality. These rules tend to result in a more efficient usage of available cache memory and read ahead mechanisms of these devices. We use these strategies in the simulation system proposed in the present paper.

This paper addresses different issues related to the problem of implementing a particle system on the GPU in a multithread architecture. The relevance and importance of transferring some of the physics computation to the GPU is in the fact that it makes possible to take out part of the load of the CPU, allowing it to process some other tasks like artificial intelligence and physics simulation optimizations.

In order to solve the collision detection between the particles, a neighborhood gathering algorithm is needed. The naive approach of such algorithm has complexity of $O(n^2)$, since it has to process each particle against all the other particles. Most of the research on particle systems tries to avoid the high complexity of proximity queries by applying some form of spatial subdivision to the environment and classifying particles among the cells based on their position. To accelerate data fetching in a parallel hardware (such as GPUs) the particles' list must be sorted in a way that all particles on the same cells are grouped together. This approach helps lowering the number of proximity queries but is very sensible to the maximum number particles that can fit in a single cell. In this work instead of using a similar approach, we propose a novel simulation architecture that maintains the particles into another kind of proximity based data structure, which we call neighborhood grid. In this data structure, each cell now fits only one particle and does not directly represent a discrete spatial subdivision. The neighborhood grid is an approximate representation of the system of neighborhoods of the environment that maps the N-dimensional environment to a discrete map (lattice) with N dimensions, so that particles that are close in a neighborhood sense appear close to each other in the map. Another approach is to think of it as a multi-dimensional compression of the environment that still keeps the original position information of all particles.

The particles are simulated and sorted as Cellular Automata with Extended Moore Neighborhood [Sarkar 2000] over the neighborhood grid, which is an ideal case for the memory model of GPUs. We argue and show that this approximate simulation technique brings a new bound to particle simulation performance, maintaining the believability for entertainment contexts. The high performance and scalability are achieved by a very low parallel complexity of the model.

To illustrate and evaluate the architecture, we implement a particle system that has a speedup of up to 9.5 over the tradition spatial hashing methods [nVidia 2007b; Microsoft 2007; Kipfer et al. 2004] , with similar visual experience. The architecture can be further extended to any other simulation model that rely on dynamic autonomous entities and neighborhood information.

Paper summary: The remainder of this paper is organized as follows. Section 2 presents a set of GPGPU concepts. Section 3 presents some related works on GPGPU that can be applied to game physics and we describe the neighborhood grid data structure, in Section 3. The particle system used in this work is presented in Section 5. In Section 4 we describe our novel simulation architecture on GPUs and acceleration data structures employed in the simulations. In Section 5 we show the results and, in Section 6, the conclusions and some considerations are presented.

2 Related Work

There are a lot of works that deals with the GPGPU field, but the application of these works on game fields are mostly concentrated on the game physics, which particle systems are a simplification of it. Physics on the GPU is a potential field and many works achieve considerable speedup by taking the physics calculations from the CPU and processing on the GPU. All the major physics engines for games available in the market had made, or are making, attempts to use of the GPU to process its calculations.

The work of Green [Green 2007] presents an implementation on the GPU of some methods of the commercial physics engine called Havok FX, which was being constructed to be a GPGPU version of Havok Physis [Havok 2009]. The Havok FX was discontinued when Havok was bought by Intel, but there has been some physics presentation with the use of OpenCL [Hensley et al. 2010]. Several other examples can be found in the literature. PhysX of nVidia [nVidia 2009b] is a physics engine that uses the CUDA architecture to optimize its calculation [Harris 2009]. Bullet [Coumans 2009], an open source physics engine, is also investing in porting it to the

GPU and has release some demos with some aspects of the engine running on the GPU. Also in [Joselli et al. 2009] a hybrid physics engine that has some of its calculations on the GPU is present.

Physics simulation works very well on the GPU because of the high performance of the stream processors, which allows high parallelism of the physics problems that can be solved in this structure. With that, it is possible to have faster physics simulation on games, and also more physics realistic games.

All the works using rigid body dynamic, similar to this work, uses some form of spatial subdivision in order to optimize the neighborhood gathering for the calculation of the collision detection. By using the spatial hash to classify the bodies into a grid, the proximity query algorithm can be performed against a reduced number of pairs. For each particle, only those inside the same grid cell and at adjacent ones, depending on its position, were considered. This strategy leads to a sequential complexity that is closer to $O(n)$. This complexity, however, is highly dependent on the maximum density of each grid cell, which can be very high if the simulated environment is large and dense. We remark that the complexity of our neighborhood grid is not affected by the size of the environment or the distribution of the particles over it. This neighborhood grid has already been used in a crowd simulation scenario with great success [Passos et al. 2010] (a minimum speed up of four times over the traditional spatial hashing).

3 Neighborhood Grid

The proposed data structure was developed to be used with a GPGPU architecture, based on CUDA [nVidia 2007c] and OpenCL [Group 2009], and, in order to keep the processing entirely at the GPU, all information about particles are mapped as textures for the display-list and vertex shader rendering. This information is stored in 3D arrays (the neighborhood grids), where each position holds the entire data for an individual particle. In this data structure, each cell fits only one particle. Figure 1 illustrates how a randomly distributed set of particle would be arranged in the neighborhood grid when correctly sorted. The smaller circles represent particles that are further away from the viewpoint.

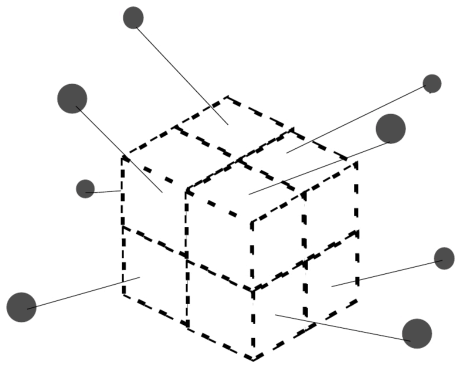

Figure 1: *An example of a distribution of the particles in the neighborhood grid. Small circles illustrate particles that are further away from the viewpoint.*

In this work we use a neighborhood gathering algorithm known as *Extended Moore Neighborhood* [Sarkar 2000] that is used in the Cellular Automata theory. Figure 2 illustrates this structure using a 2D matrix holding arbitrary information for 36 individual particles. To reduce the cost of proximity queries, each particle only gather information about the particles surrounding its cell, based on a constant search radius. In the example of Figure 2, we chose the radius to be 2, so the particle entity represented at cell (2,2) (in light gray) would have access to the 24 highlighted surrounding cells (represented in dark gray).

This kind of spatial data structure with extremely regular information gathering enables a good prediction of the performance, since the number of proximity queries will always be constant over the simulation. This happens because instead of making these proximity queries over all particles inside a coarse grid bucket/cell (variable quantity), such as in traditional implementations, each particle would query only a fixed number of surrounding individual neighbors. However, this grid/matrix has to be sorted continually in such a way that those entities, which are neighbors in geometric space are stored in individual cells that are close in the neighborhood grid. This guarantees that each particle should gather information only about its closest neighbors. During the simulation (and depending on the sorting step), some misalignment may occur over the data structure causing that some of the neighbor particles are missed by the gathering step. However, the larger the gathering radius is, less likely it is to happen such issue, as we will observe latter when we present our experiments.

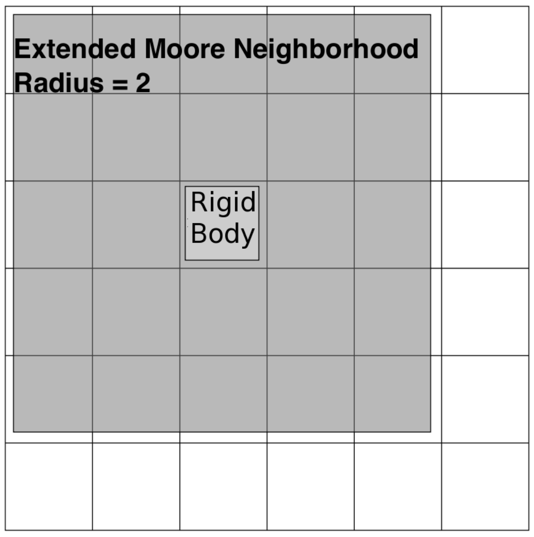

Figure 2: *Example of the Structure of the Extended Moore Neighborhood with 36 particles and radius = 2.*

Since in this work the simulation of each particle is mapped to one GPGPU thread for both the sorting and simulation steps, it is important to mention that the grids are double buffered; consequently each of these tasks does not write data over the input structures that can still be read by the other parallel GPGPU threads. This work could also use atomic operations for the grid operations, but these kind of operation is still very costly for massive simulations.

The position information of each particle is used to perform a lexicographical sort based on the three dimensions coordinate of the vector. The goal is to store in the closer-bottom-left cell of the grid the particle with the smaller values for Z, Y and X, and in the far-top-right cell the particle with highest values of Z, Y and X respectively. Using these three values to sort the grid, the farthest lines will be filled with the rigid particle with the higher values of Z while the top lines will be filled with the particles with higher values of Y and the right columns will store those with higher values for X and so on. This kind of sorting provides data for the approximate neighborhood query, which is optimal in terms of data locality.

Our proposed architecture is independent of the sorting algorithm used, as long as the rules above are always, eventually or even partially achieved during simulation, depending on the desired neighborhood precision. In this work we use a bitonic sort [Batcher 1968], which makes a full sort in each dimension.

The bitonic sort [Batcher 1968] is a simple parallel sorting algorithm that is very efficient when sorting small number of elements [Blelloch et al. 1998], which is our case since our sort strategy is divided by dimensions. Our implementation is an optimized and adapted version based on a previous work of nVidia [nVidia 2007a]. This sort is divided in 3 passes, one for each dimension (X,Y and Z).

4 Architecture environment

Our proposed architecture implements a particle system using a novel GPU computing solution, based at the neighborhood grid data structure, allowing a high performance increase during simulation.

The proposed architecture is based on particle systems like the ones presented in [nVidia 2007b; Microsoft 2007; Kipfer et al. 2004] and in a hybrid physics engine [Joselli et al. 2009].

The loop of the proposed physics engine is responsible for: Detecting collisions in two phases: a broad phase (with the neighborhood grid) and a narrow phase; Applying the external forces on the particles, like the gravity force; Forwarding the simulation step for each particle by computing the new position and velocities according to the forces and the time step, i.e., integrating the equations of motion.

At the beginning, the architecture sort all particles according to its position in the 2 or 3 axis, depending if its a grid or matrix. The sorted particles gather its neighborhoods according to the radius and calculate its collisions. Based on these results, the system calculates the result forces at the particles and add any external forces that may be influencing it, such as gravity and users' input. Finally the system calculates the new velocity and positions, integrating the whole system.

The proposed architecture is built in a way such that it can be used, with proper modifications, with games. It was implemented using the following technology: CUDA [nVidia 2009a] for GPGPU processing; OpenGL for rendering; GLSL (OpenGL Shading Language) for shaders; and GLUT (OpenGL Utility Toolkit) for window creation and input gathering. But the concepts presented here could also be adapted to others technologies.

The data that is exchanged between the CPU and GPU is encapsulate in a special structure, in order to keep the communication between the CPU and the GPU to a minimum, since this process can be a bottleneck of any simulation that has communication between CPU and GPU [Krueger 2008].

Neighborhood Gathering or the broad phase of the collision detection: Each of the particles needs to find its neighborhood bodies for calculating its collisions. This operation has complexity of $O(n^2)$ for a collection of n particles using a brute force method, which is very time consuming, even for a small set of particles. To avoid this time complexity, this paper employs the neighborhood grid/matrix data structure presented in Section 3.

The narrow phase of the collision detection: The narrow phase of the collision detection is responsible for doing the actually collision detection among the particles. In this work, instead of doing the collision check between all the polygons of the entities, it is implemented a basic primitive area element, that complex models are put inside.

The bounds are used to surround every model, simplifying the narrow phase of the collision detection. Two types of bounds were implemented: a circle bound and a bounding rectangle.

Integration: After fluid's forces are computed, it's necessary to integrate the particles velocity and position. This method is responsible for integrating the equations of motion of a particle [Eberly 2004]. This method updates the particle velocity based on the forces that are applied to it, which are sent to the integrator, and then it updates the position based on its velocities, using a method based on Euler integration (this approach is one of the simplest forms of integration) using a finite time step.

5 Results

This section presents the results obtained from our proposed architecture. We used a PC equipped with an Intel Core 2 Duo 2.66GHz using 2 GB of RAM and a NVidia GTX480. Simulations tests with different configurations were performed. . An screenshot of the simulation can be seen in Figure 3.

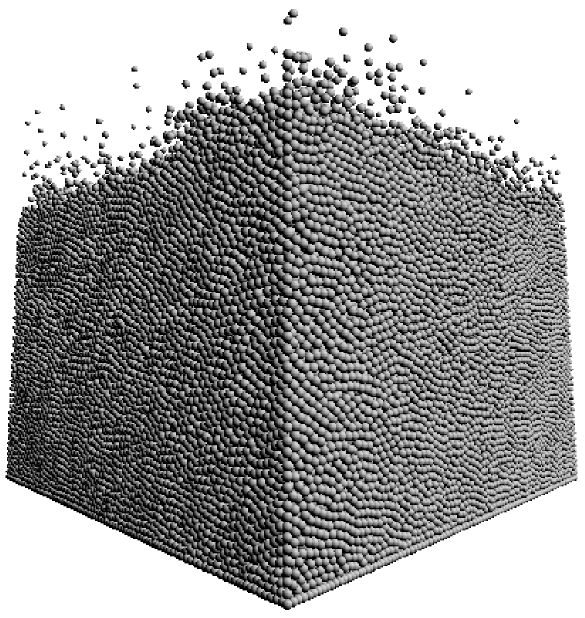

Figure 3: *Screenshot of the Simulation.*

To evaluate the scalability of the architecture, we varied the number of particles being simulated (from 1 thousands to 262 thousands) with the Moore neighborhood radius set to 4. In order to full evaluate the speedup of this architecture for particles systems over the traditional spatial hashing method, we have implemented the spatial hashing scheme in GPU with the use of CUDA.

In Table 1 we present the results of different simulation configurations, varying the number of particles using the neighborhood grid method and the spatial hashing method. In both methods, all processing is done in the GPU. The time was take by the architecture to process and render one frame of the application in miliseconds. *Speedup* is defined by the relation $S = \frac{X_1}{Y_2}$, being X_1 the time for the Neighborhood Grid and Y_2 the time for the Spatial Hash. As expected, the simulation using our neighborhood grid method presents the better result than the simulation using the traditional spatial hashing.

Table 1: *Scalability of the Simulation when using the Spatial Hash and the Neighborhood Grid.*

Particles	Uniform Grid	Neighborhood Grid	Speedup
1,024	0.426	0.419	1.18
2,048	0.450	0.435	1.03
4,096	0.559	0.539	1.04
8,192	0.835	0.743	1.12
16,384	1.01	0.892	1.13
32,768	1.631	1.305	1.25
65,536	4.608	2.092	2.20
131,072	17.54	3.636	4.82
262,144	83.33	8.772	9.5

Figure 4 shows the same results of table 1 in a graphic. From this, we can better see that our architecture is faster for particles systems and scales better than the spatial hashing.

6 Conclusion and Future works

In this paper we have shown an extension of a novel technique for simulating particles systems in real time using the GPU. This architecture is capable of interactively simulating and rendering up to 200,00 bodies in real time frame rate, while the traditional spatial hashing methods barely maintains interactivity in the simulation. Moreover, with our architecture and bitonic sort, configured with a radius of 4, we experience a similar visual simulation as with the spatial hashing method with expressive speedup. The authors

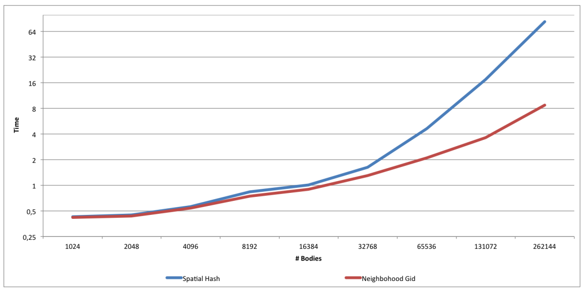

Figure 4: *Evolution of the Simulation in milliseconds in a log2 scale.*

of this work suggest using this configuration to achieve best visual and performance for simulating particle systems.

As presented by the results, performing particle system simulation using the neighborhood grid method increases simulation performance, obtaining a speedup of more than 9 times over the traditional spatial hashing simulation.

References

BATCHER, K. E. 1968. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, 307–314.

BLELLOCH, G. E., PLAXTON, C. G., LEISERSON, C. E., SMITH, S. J., MAGGS, B. M., AND ZAGHA, M. 1998. An experimental analysis of parallel sorting algorithms. Tech. rep.

COUMANS, E., 2009. Bullet physics library. Disponvel em: `http://www.bulletphysics.com`.

EBERLY, D. H. 2004. *Game Physics*. Morgan Kaufmann.

GEORGII, J., ECHTLER, F., AND WESTERMANN, R. 2005. Interactive simulation of deformable bodies on gpu. In *Proceedings of Simulation and Visualization 2005*, 247–258.

GOVINDARAJU, K. N., REDON, S., LIN, M. C., AND MANOCHA, D. 2003. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware 2003*, 25–32.

GREEN, S., 2007. Gpgpu physics. Siggraph07 GPGPU Tutorial.

GROUP, K. 2009. Opencl - the open standard for parallel programming of heterogeneous systems. Tech. rep., Avalible at: `http://www.khronos.org/opencl/`. 20/12/2010.

HARRIS, M., 2009. Cuda fluid simulation in nvidia physx. Siggraph Asia 2009: Beyond Programmable Shading course.

HAVOK, 2009. Havok physics. Avalible at: `http://www.havok.com/content/view/17/30/`.

HENSLEY, J., GERSTMANN, D., AND YANG, J. 2010. Physical and graphical effects in opencl by example: (copyright restrictions prevent acm from providing the full text for this article). In *ACM SIGGRAPH ASIA 2010 Courses*, ACM, New York, NY, USA, SA '10, 11:1–11:1.

JOSELLI, M., CLUA, E., MONTENEGRO, A., CONCI, A., AND PAGLIOSA, P. 2008. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 149–156.

JOSELLI, M., ZAMITH, M., CLUA, E., MONTENEGRO, A., LEAL-TOLEDO, R., CONCI, A., PAGLIOSA, P., VALENTE, L., AND FEIJÓ, B. 2009. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Comput. Entertain. 7*, 4, 1–15.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: a gpu-based particle engine. In *Graphics Hardware 2004*, 115–122.

KRUEGER, J. 2008. A gpu framework for interactive simulation and rendering of fluid effects. *IT - Information Technology 4*, (accepted).

MICROSOFT, 2007. Advanced particles. Siggraph 2007: Real-Time Rendering in 3D Graphics and Games course.

NVIDIA. 2007. Bitonic sort demo. Tech. rep., Avalible at: `http://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_ Algorithms.html#bitonic`.

NVIDIA. 2007. Cuda particles. Tech. rep., Avalible at: `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/particles/doc/particles.pdf`. 20/02/2008.

NVIDIA. 2007. Nvidia cuda compute unified device architecture documentation version 1.1. Tech. rep., Avalible at: `http://developer.nvidia.com/object/cuda.html`. 20/12/2007.

NVIDIA, 2009. Nvidia cuda compute unified device architecture documentation version 2.2. Avalible at: `http://developer.nvidia.com/object/cuda.html`.

NVIDIA, 2009. Physx. Avalible at: `http://www.nvidia.com/object/nvidia_physx.html`. 20/02/2009.

PASSOS, E. B., JOSELLI, M., ZAMITH, M., CLUA, E. W. G., MONTENEGRO, A., CONCI, A., AND FEIJO, B. 2010. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. *Comput. Entertain. 7* (January), 60:1–60:15.

SARKAR, P. 2000. A brief history of cellular automata. *ACM Comput. Surv. 32*, 1, 80–107.