



# CUDA Particles

Simon Green  
[sdkfeedback@nvidia.com](mailto:sdkfeedback@nvidia.com)

---

June 2008

# Document Change History

<b>Version</b>	<b>Date</b>	<b>Responsible</b>	<b>Reason for Change</b>
1.0	Sept 19 2007	Simon Green	Initial draft
1.1	Nov 3 2007	Simon Green	Fixed some mistakes, added detail.
1.2	June 10 2007	Simon Green	Updated to match code in CUDA 2.0 release

# Abstract

Particle systems [1] are a commonly used technique for simulating physical systems. In this document we will describe how to efficiently implement a particle system in CUDA, including interactions between particles using a uniform grid data structure.

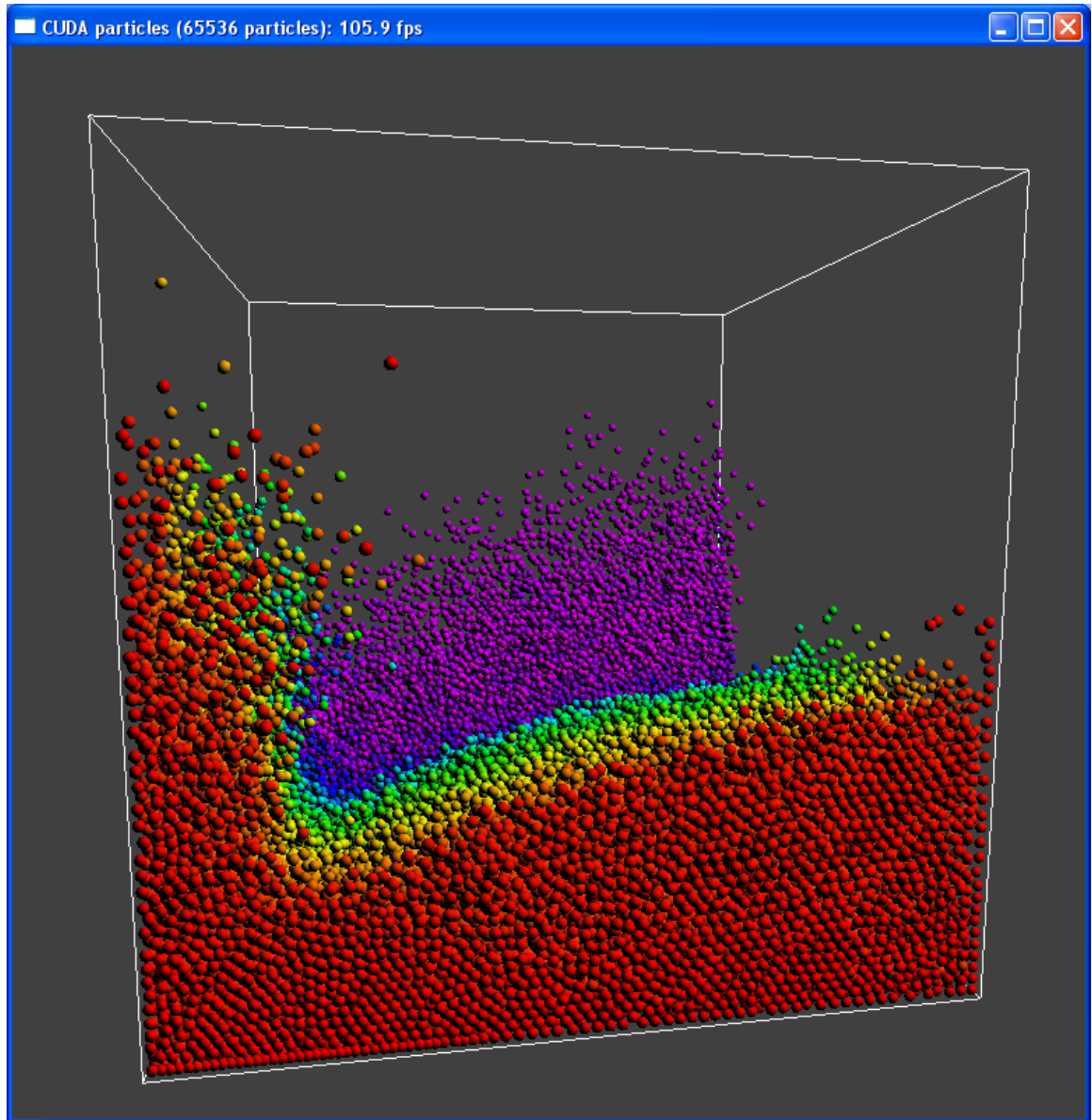


Figure 1. Particle simulation in CUDA.

# Introduction

Particle-based techniques are used in many applications - from interactive simulation of fluids and smoke for games to astrophysics simulations and molecular dynamics. Recent research has also applied particle methods to soft body and cloth simulation [4], and there is some hope that one day these techniques will allow an efficient unification of rigid, soft body and fluid simulations where everything can interact with everything else seamlessly.

There are two basic types of simulation – Eulerian (grid-based) methods, which calculate the properties of the simulation at a set of fixed points in space, and Lagrangian (particle) methods, which calculate the properties of a set of particles as they move through space.

Particle-based methods have several advantages over grid-based methods.

- They only perform computation where necessary.
- They require less storage and bandwidth since the model properties are only stored at the particle positions, rather than at every point in space.
- They are not necessarily constrained to a finite box.
- Conservation of mass is simple (since each particle represents a fixed amount of mass).

The main disadvantage of particle-based methods is that they require a very large number of particles to obtain realistic results. Fortunately, it is relatively easy to parallelize particle systems and the massive parallel computation capabilities of modern GPUs now makes it possible to simulate large systems at interactive rates.

Note that some techniques (such as the particle level-set method) attempt to combine the strengths of both methods.

This document describes how to implement a simple particle system in CUDA, including particle collisions using a uniform grid data structure.

The accompanying code is intended to provide a framework to which more complicated particle interactions such as smoothed particle hydrodynamics [2, 3] or soft body simulation can be added.

# Demo Usage

Press 'v' to enter view mode. In view mode:

- Hold down the left mouse button to rotate the camera.
- Hold down the middle mouse button to translate the camera.
- Hold down the left and middle buttons and move up and down to zoom.

Press 'm' to change to move mode, where you can move the collision sphere to interact with the particles.

Press '1' or '2' to reset the particles. '3' drops a ball of particles into the system.

Press 'h' to display the sliders, which enables you to modify the simulation parameters interactively.

For an interesting effect, try turning the gravity to zero, and the collision attraction to 0.1. This will cause the particles to group together like a sticky substance.

If you have a compute capability 1.1 GPU, you can enable the atomic processing path by editing the file "particles\_kernel.cuh" so that `USE_SORT` is set to 0, and changing the custom build setup on "particleSystem.cu" so that `nvcc` is called with "`-arch=sm_11`".

# Implementation

There are three main steps to the performing the simulation:

1. Integration.
2. Building the grid data structure
3. Processing collisions

Rendering of the particles is performed using OpenGL, making use of point sprites and a GLSL pixel shader that makes the points appear spherical.

---

## Integration

The integration step is the simplest step. It integrates the particle attributes (position and velocity) to move the particles through space. We use Euler integration for simplicity - the velocity is updated based on applied forces and gravity, and then the position is updated based on the velocity. Damping and interactions with the bounding cube are also applied in this stage.

The particle positions and velocities are both stored in float4 arrays. The positions are actually allocated in an OpenGL vertex array object (VBO) so that they can be rendered from directly. This VBO memory is mapped for use by CUDA using `“cudaGLMapBufferObject”`. The arrays are double-buffered so that updating the new values will not affect particles not yet processed.

---

## Particle-Particle Interactions

It is relatively simple to implement a particle system where particles do not interact with each other. Most particle systems used in games today fall into this category. In this case each particle is independent and they can be simulated trivially in parallel.

The “nbody” sample included in the CUDA SDK includes interactions in the form of gravitational attraction between bodies. It demonstrates that it is possible to get excellent performance for n-body gravitational simulation using CUDA when performing the interaction calculations in a brute-force manner – computing all  $n^2$  interactions for n bodies. The use of shared memory means that this method does not become bound by memory bandwidth.

However, for local interactions (such as collisions) we can improve performance by using spatial subdivision.

The key insight here is that for many types of interaction, the interaction force drops off with distance. This means that we can compute the force for a given particle by only comparing it with all its neighbors within a certain radius.

Spatial subdivision techniques divide the simulation space so that it is easier to find the neighbors of a given particle.

---

## Uniform Grid

In this sample we use a uniform grid, which is the simplest possible spatial subdivision. (It is possible this technique could be extended to more sophisticated structures such as hierarchical grids, but we don't discuss this here).

A uniform grid subdivides the simulation space into a grid of uniformly sized cells. For simplicity, we use a grid where the cell size is the same as the size of the particle (double its radius). This means that each particle can cover only a limited number of grid cells (8 in 3 dimensions). Also, if we assume no inter-penetration between particles, there is a fixed upper bound on the number of particles per grid cell (4 in 3 dimensions).

Each particle is assigned to a grid cell based on its center point. The index of the particle is stored in the cell.

The grid data structure is generated from scratch each time step. It may be possible to perform incremental updates to the grid structure on the GPU, but this method is simple and the performance is constant regardless of the movement of the particles.

We implement two different methods for generating the grid structure depending on the capability of the GPU.

### Building the Grid using Atomic Operations

On GPUs that support atomic operations (compute capability 1.1), building the grid is relatively simple.

Atomic operations allow multiple threads to update the same value in global memory simultaneously without conflicts.

We use 2 arrays in global memory:

`gridCounters` – this stores the number of particles in each cell so far. It is initialized to zero at the start of each frame.

`gridCells` – this stores the particle indices for each cell, and has room for a fixed maximum number of particles per cell.

The “`updateGrid`” kernel function updates the grid structure. It runs with one thread per particle. Each particle calculates which grid cell it is in, and uses the `atomicAdd` function to atomically increment the cell counter corresponding to this location. It then writes its index into the grid array at the correct position (using a scattered global write).

Figure 2 shows a simple example with 6 particles in a 2D grid.

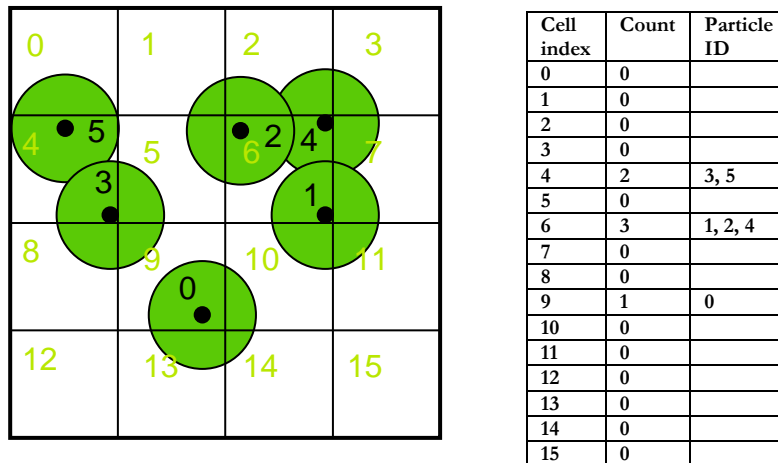


Figure 2 – Uniform Grid using Atomics

Note that the global memory writes in this pass are essentially random (depending on the position of the particles), and so will not be coalesced. In addition, if multiple particles write to the same cell location simultaneously, the writes will be serialized, causing further performance degradation. Despite these factors, the grid generation is not a major bottleneck compared to the collision processing.

## Building the Grid using Sorting

For GPUs such as the GeForce 8880 which do not support atomic operations, we need to use a more complex solution based on sorting.

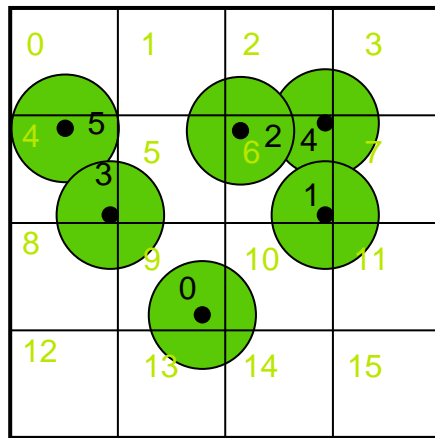
The algorithm consists of several kernels. The first kernel “`calcHash`” calculates a hash value for each particle based on its cell id. In this example we simply use the linear cell id as the hash, but it may be beneficial to use other functions such the Z-order curve [8] to improve the coherence of memory accesses. The kernel stores the results to the “`particleHash`” array in global memory as a `uint2` pair (cell hash, particle id).

We then sort the particles based on their hash values. The sorting is performed using a fast radix sort as described by Scott Le Grand in his article in GPU Gems 3 [6]. This creates a list of particle ids in cell order.

In order for this sorted list to be useful, we need to be able to find the start of any given cell in the sorted list. This is achieved by running another kernel “`findCellStart`”, which uses a thread per particle and compares the cell index of the current particle with the cell index of the previous particle in the sorted list. If the index is different, this indicates the start of a new cell, and the start address is written to another array using a scattered write. Note that this was not possible on previous architectures because of the lack of scattered write support and a binary search would have to be used instead [9].

Figure 3 demonstrates creating the grid using the sorting method.





Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	
6			2
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figure 3 – Uniform Grid using Sorting

## Interactions

Once we have built the grid structure we can use it to accelerate particle-particle interactions. In the sample code we perform simple collisions between particles using the DEM method [5]. This collision model consists of several forces, including a spring force which forces the particles apart, and a dashpot force which causes damping.

Each particle calculates which grid cell it is in. It then loops over the neighboring 27 grid cells (3x3x3 cells) and checks for collisions with each of the particles in these cells. If there is a collision the particle's velocity is modified.

For the sort-based method, we can tell when we have reached the end of the current cell when the current cell index no longer equals the index of the cell we are examining.

## Optimization

Since the particles can move to any position in the grid, the memory reads for the position and velocity of the neighboring particles will be at effectively random locations. If we use global memory loads to obtain this data, they will always be un-coalesced. For this reason we bind the global memory arrays to textures and use texture lookups (`tex1Dfetch`) instead, which improves performance by up to 45% since texture reads are cached.

As an additional optimization when using the sorting method, we actually re-order the position and velocity arrays into sorted order to improve the coherence of the texture lookups during the collision processing stage.

The code included in the CUDA SDK can simulate 65,536 colliding particles at about 120 frames per second (fps) in the steady state on a GeForce 8800 GTS. This is using the sort-based algorithm, which is currently the fastest method, largely because the sorting improves the coherence of memory access.

Although simpler to implement, the atomic-based algorithm is generally slower, at around 83 fps for 65536 particles. It is also more sensitive to the distribution of the particles, with performance dropping to 45 fps for a random initial distribution. It would be possible to

sort the particles in the atomic method also, but then you might as well use sorting for the bucketing as well.

## Conclusion

The ability of CUDA to perform scattered memory writes makes it possible to build dynamic data structures on the GPU. Sorting can be used to improve memory coherence when accessing these kind of data structures. This combined with the computational power of the GPU makes it possible to simulate large systems of interacting particles at interactive rates.

The code included in this sample is by no means optimal and there are many possible further optimizations to this algorithm [see 10 for details]. In future versions of this sample we would like to try processing collisions on a per-cell basis, which would allow using shared memory to cache neighboring particles and hopefully enable even higher performance. It would also be interesting to try sorting particles in different orders to maximize memory coherence.

# Bibliography

1. Reeves, W. T. 1983. Particle Systems—a Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Graph.* 2, 2 (Apr. 1983), 91-108.
2. Monaghan J.: Smoothed particle hydrodynamics. *Annu. Rev. Astron. Physics* 30 (1992), 543. 12, 13
3. Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. *Proceedings of 2003 ACM SIGGRAPH Symposium on Computer Animation* (2003), 154–159.
4. Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. 2007. Position based dynamics. *J. Vis. Commun. Image Represent.* 18, 2 (Apr. 2007), 109-118.
5. Harada, T.: *Real-Time Rigid Body Simulation on GPUs*. GPU Gems 3. Addison Wesley, 2007
6. Le Grand, S.: *Broad-Phase Collision Detection with CUDA*. GPU Gems 3, Addison Wesley, 2007
7. Nyland, L., Harris, M., Prins, J.: *Fast N-Body Simulation with CUDA*. GPU Gems 3. Addison Wesley, 2007
8. Z-order (curve), Wikipedia [http://en.wikipedia.org/wiki/Z-order\\_\(curve\)](http://en.wikipedia.org/wiki/Z-order_(curve))
9. Ian Buck and Tim Purcell, *A Toolkit for Computation on GPUs*, GPU Gems, Addison-Wesley, 2004
10. Qiming Hou, Kun Zhou, Baining Guo, [BSGP: Bulk-Synchronous GPU Programming](#), ACM TOG (SIGGRAPH 2008)



#### Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

#### Trademarks

NVIDIA, the NVIDIA logo, GeForce, NVIDIA Quadro, and NVIDIA CUDA are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

#### Copyright

© 2007 NVIDIA Corporation. All rights reserved.

