GPU bucket sort algorithm with applications to nearest-neighbour search

T. Rożen, K. Boryczko, W. Alda
AGH University of Science and Technology,
Institute of Computer Science,
al. Mickiewicza 30, 30-059 Kraków, Poland
{rozen, boryczko, alda}@agh.edu.pl

ABSTRACT

We present an adoption of the bucket sort algorithm capable of running entirely on GPU architecture. Our implementation employs render-to-texture to enable scatter operation. Linked lists of elements in each bucket are build and stored directly in video memory. We show also the use of this sorting method in a particle-based simulation. Dissipative Particle Dynamics is the physical model of choice; the simulation is performed entirely on the graphics hardware. GPU bucket sorting is used to build nearest-neighbour maps on a regular cell-grid which are the input of interparticle interaction computation. Finally we implement a simple random-number generator which is required by the DPD method.

Keywords: Computer graphics and animation, GPU programming, Nearest-neighbour search algorithm, Fluid simulation

1 INTRODUCTION

One of the fundamental problems in computer simulation of molecular and particle dynamics (e.g. Dissipative Particle Dynamics [HK92]) is the determination of interacting atoms or molecules. In order to compute all forces acting on a single particle a set of interacting neighbours in a given proximity has to be found.

The straightforward approach of linear scanning through all particles cannot be applied to any but the simplest simulation environment. Several solutions have been proposed to alleviate the problem. Most of them are based on subdividing the simulation space and reducing the number of searched elements [AMN+98], e.g. kd-trees [Ben75], well suited for unstructured random data. However, when the particles are almost uniformly distributed in space a simple yet effective method for nearest neighbour search is to distribute them into a regular cell grid and to look for neighbours in spatially close cells. This is the case of fluid simulation with low compressibility where each particle has roughly the same number of neighbours.

Particle-based simulation methods require significant computational power. Through recent years we have witnessed a growing interest in using commodity graphics hardware in general-purpose computations. This is due to its increasing performance as well as flexibility in data and instruction handling, allowing quicker solution than traditional CPU implementations to a variety of problems [OLG+05]. The problem of nearest-neighbour search is also an important issue in ray-tracing and global illumination. Purcell [Pur04] has demonstrated the use of sorting on a GPU by building a regular cell-grid where a single cell's size equals the search radius and then sorting particles based on the cell number.

Purcell used bitonic merge sort algorithm [PDC+03] to order photons by cells. Bitonic merge sort is based on a sorting network [LKO05]. It doesn't require arbitrary write operation thus allowing straightforward implementation on GPU. Moreover it always executes the same sequence of steps regardless of the input data. The downside is the computational complexity, which is $\Theta(nlog^2n)$. Cache usage improvements to this algorithm have been introduced by Govindaraju et al. [GRHM05]. Their memory usage pattern reduces bandwith overhead and allows for optimal throughput resulting in faster sorting times, however the algorithmic complexity has not been reduced. A recent improvement to GPU sorting by Gress and Zachman [GZ06a] based on adaptive bitonic sorting achieves optimal complexity of $O(nlogn)$. However using any of the above mentioned techniques requires further post-processing with binary search to find a range of neighbours.

Alternative method for particle-based simulation has been proposed by Amada et al. [AIY+04] where a neighbourhood map is pre-computed on the CPU and then at each step transfered to graphics memory to be used during simulation. Also KD-tree methods have been successfully implemented on the graphics hardware by Foley and Sugerman [FS05].

Particle simulation methods which do not require an explicit nearest-heighbour search have also been investigated, e.g. by Kolb and Cuntz [KC05] who used force accumulation on a 3D grid to solve Smoothed Particle Hydrodynamics equations. A similar approach has been used by Mueller et al. [MCG03]. SPH-based simulation has been the foundation of work by Hegeman et al. [HCM06], however they computed exact interparticle interactions by using a dynamic quad-tree to find particle neighbours. An interesting contribution is the paper by Kipfer et al. [KSW04] who have build a particle engine for simulating and rendering large particle sets on the GPU. Interparticle collisions are approximated by finding a set of potential colliders in a 2D texture. Nearest-neighbour search on the GPU has also been investigated by Bustos et al. in [BDH$^+$06] the context of database operations.

A recent paper by Harada et al. [HKK07] shows implementation of limited bucket sorting in Smoothed Particles Hydrodynamics simulation. Their method allows for a maximum of four particle references in a single grid cell.

We present a novel GPU bucket sorting algorithm that builds linked lists of neighbours from regular cell-grid with application to nearest neighbour search in particle based simulation.

2 BUCKET SORT ALGORITHM FOR THE GPU

Bucket sort [CLR89] algorithm is a sorting algorithm that runs in linear time. It works by partitioning the problem domain into a finite number of buckets and assigning each element to a bucket. The process may be repeated recursively or another algorithm may be used to further sort elements in each bucket. For many applications however (e.g. nearest neighbour search, see next section) it may be sufficient just to distribute elements to buckets. The classical bucket sorting achieves $\Theta(n)$ complexity by scanning only once through the input data and inserting the elements into lists corresponding to each bucket. This behaviour cannot be easily reproduced on the GPU due to the limitations in scatter operation. In this chapter we present a modified bucket sort algorithm that can be successfully implemented on the resource limited hardware.

Listing 1 shows pseudo code of our algorithm. N and M parameters are the numbers of element and bucket count respectively. The array $a[N]$ holds the bucket identifiers to which the array elements will be put, i.e. element i will be placed in the bucket pointed to by $a[i]$. When the algorithm stops two arrays are returned, $head[M]$ and $next[N]$, which make up a linked list of elements in each bucket. The first holds identifiers of the first element in each bucket while the second one points to the next element in the same bucket. A spe-

Listing 1: GPU bucket sort algorithm

```
bucket_sort(a[N], head[M], next[N])
1   fill(head, NULL)
2   fill(next, NULL)
3   fill(visited, false)
4   while true
5     finished = true
6     for i = 0 to N-1
7       if not visited[i]
8         head[a[i]] = i
9         finished = false
10    if finished
11      break
12    for i = 0 to N-1
13      if not visited[i]
14        if head[a[i]] == i
15          visited[i] = true
16        else
17          next[i] = head[a[i]]
```

cial *NULL* value is put at the end of each list (or into an empty bucket).

The algorithm performs bucket sorting by employing two simple steps in a loop: (i) the elements, which have not yet been inserted into any bucket, are put into lists' heads and then (ii) all items have their next pointers set to the head element in their buckets (except for the elements that are currently at the head of bucket's list). Additionally step (ii) marks the head elements as computed (*visited*[N] array) thus leaving them out from the following iterations. The loop terminates when all elements have been assigned to a list.

Figure 1 visualises the way our algorithm works for a simple case with eight elements distributed into four buckets.

The average computational complexity of the proposed algorithm is $O(N^2/M)$. In practical case with a large number of buckets and uniformly distributed items in the buckets this algorithm can perform very well. However, the lower bound for the running time is $\Theta(N^2)$, which may happen if all elements are to be put in the same bucket. As mentioned before in our application to fluid simulation this will never be the case thanks to particle repulsion.

2.1 GPU implementation details

The algorithm presented in pseudo-code listing has been implemented on modern graphics hardware using OpenGL API, mapped to the following steps:

1. Initialise textures

2. Create points from elements (vertex buffer)
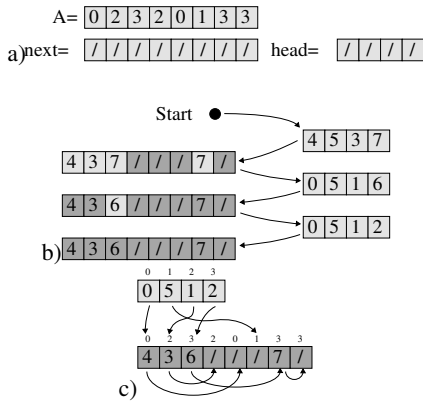
3. Scatter elements to buckets heads

Figure 1: Algorithm execution example a) content of input data: *A* input array with element cell numbers [0..7] (constant through execution time), *head* first element in each bucket, *next* next element in the same bucket; a special value "/" (*NULL*) indicates the end of a linked list b) execution steps; dark grey elements of *next* array have their visited flag set (excluded from further computation) c) resulting data.
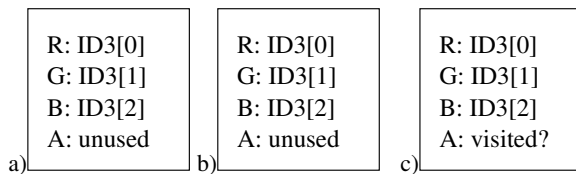

Figure 2: Data representation using texture memory: a) mapping texture: element bucket mapping (3ID) b) head texture: first element in each bucket (3ID) c) next texture: next element in the same bucket (3ID) and a visited flag.

4. Occlusion query of step 2: STOP if nothing drawn

5. Update next pointers

6. Mark elements in buckets' heads as visited

7. Repeat from step 2

First, texture memory is initialised for data structures (see Figure 2). RGBA texture format is used with 8-bit precision per channel. Bucket and elements identifiers are encoded using 3 bytes into R, G and B colour channel respectively. A special value of $(255, 255, 255)$ indicates the end of a linked list or empty bucket (NULL value). This scheme allows for effective number of 16777215 item identifiers. The mapping texture is initialised with user provided data while head and next textures are initially filled with *NULLs*.

A simple scatter operation is employed in order to assign elements to buckets heads. Mapping texture is copied to a vertex buffer, using copy-to-vertex buffer OpenGL extension GL_ARB_pixel_buffer_object, and then used to render points on the head texture. If multiple points end up in the same bucket, all except one will be overwritten and the same operation will be repeated

for the remaining points. During point rendering occlusion is queried to count drawn points. If it is zero the algorithm is stopped.

At the end of each iteration next pointers of unvisited elements are updated and elements currently in bucket head are marked as visited.

A simple optimisation has been added to the scatter step to reduce the amount of overwritten buckets. The vertex buffer with elements is divided into a number of equal parts, each has an occlusion object attached. During the initial step only the first sub-vertex buffer is used. In subsequent steps other sub-buffers are added to rendering only if the previous one's not visited element count reaches a predefined threshold. If a occlusion query indicates that for a sub-buffer no points are being drawn this buffer is excluded from future rendering. The algorithm stops if there are no more elements to draw in any of the vertex buffers (all queries returned zero pixels drawn). This partial-update approach allows for significant performance boost.

2.2 Re-sort algorithms

The presented algorithm has a useful property in that it allows for efficient re-sorting of input data. This is important for many applications which need to initially sort their data and then periodically update the list to accommodate changes in element order.

Our algorithm is able to re-sort the sequence by altering only the elements which are not in their destination buckets. At first, elements' lists are scanned for elements that should be moved to another buckets. Such elements are removed from the list by updating next pointers of sibling elements and clearing the visited flag for such elements. After this operation the sorting algorithm is started as described above but the number of elements that needs sorting is smaller resulting in a low number of iterations.

Partial buffer update optimisation needs to be adjusted depending on the count of elements that need re-sorting. When the number is small only a few (or even one) vertex buffer may be used.

2.3 Application to nearest neighbour search

One of the possible applications of our sorting algorithm is to the problem of nearest neighbour search. An example of such application, Dissipative Particle Dynamics simulation, is presented in the following chapter. In our sample a set of particles representing physical fluid are simulated in three dimensional space. In order to compute particle-particle interaction for each particle a set of neighbours needs to be found in a specified radius. To accomplish this the simulation space is evenly divided along each axis into cubes with edge length equal to neighbour search distance. Each cube

has a unique sequential identifier assigned. Cube numbering first goes along the *x* axis, then increases along *y* axis and finally by *z* axis.

All particles from the simulation domain are assigned to corresponding cubes based on their spatial location. Because of the cell numbering scheme introduced it's a straightforward task to compute cell number for each one. Particles are marked with their corresponding cell identifier.

Now our bucket sort algorithm is employed to sort the particles. This results in two arrays:

- there is a link between each cell and the first particle which belongs there,

- each particle has an identifier of a next one in the same cell.

These arrays form a linked list of particles for each cell.

Nearest neighbours of a particle can now be narrowed down to the particles in the current cell and neighbouring cells (27 total for 3D space). These may however include more particles than desired, so an additional distance check needs to be performed in order to get the exact neighbours set.

3 EXPERIMENTAL RESULTS

Our test environment included a GeForce 6800 graphics card with 256MB of video memory. The CPU was a Pentium IV 3.0 GHz. OpenGL 2.0 has been used as the graphics API and all pixel and vertex shaders have been implemented using GLSL. Benchmarking data comes from the particle-based simulation described in the following chapter.

The first set of tests measured the performance of bucket sort algorithm for several grid sizes and particle-per-cell numbers. Table 1 shows the obtained results. As expected computation time increases with data count. Also, when the particle count/cell number ratio is high more iterations are executed.

Re-sorting times have also been tested. This is an important issue e.g. in particle simulations where usually only a fraction of the total number changes their cell location. Table 2 shows the results for a 1048576 particle set-up, with a 64^3 grid. This test included two steps. The former consisted of bucket sorting the input data the normal way. With sorted data some particles were displaced to other cells and the re-sorting algorithm has been applied.

We have also compared our solution with another GPU sorting algorithm: GPUSort version 2.0 by Govindaraju et al. [GRHM05]. This is a general purpose sorting algorithm which produces an ordered array out of arbitrary data. To achieve the results of bucket sorting input data is first sorted by grid cell key followed by a binary search to locate the first and the last element in each cell. Such approach has been employed by Purcell
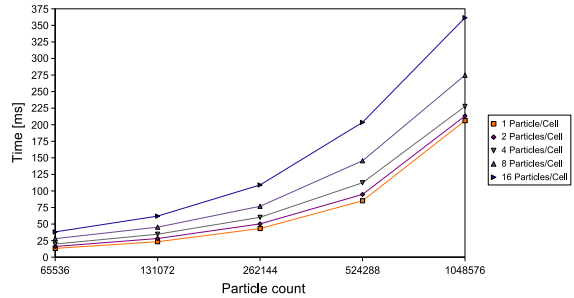
Particles	1/Cell	2/Cell	4/Cell	8/Cell	16/Cell
65536	13.28	16.15	19.84	28.01	38.17
131072	23.33	28.21	34.7	45.33	62.03
262144	43.27	50.26	60.24	76.86	109.02
524288	85.1	94.79	112.36	145.57	203.58
1048576	206.15	213.25	227.66	274.73	361.52

Table 1: Sorting times on a GeForce 6800 Ultra with different average particle to cell number ratio.
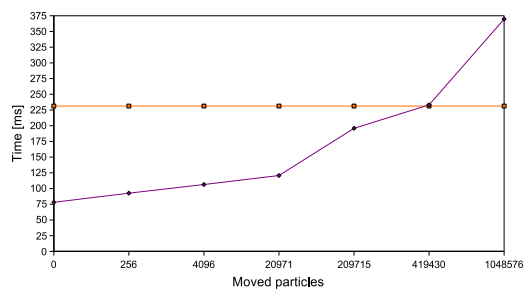
Moved particles	Sort	Re-sort
0	231.26	78.07
256	231.26	92.63
4096	231.26	106.4
20971	231.26	120.73
209715	231.26	195.87
419430	231.26	233.13
1048576	231.26	369.75

Table 2: Re-sorting times on a GeForce 6800 Ultra compared to full sorting when certain number of particles has been moved to another cell.

and Donner [PDC+03] to global illumination rendering. On the other hand our algorithm produces lists of particles for each cell so no additional step is required. Comparison results are shown in table 3. We have also included the timing of re-sorting with 2% of particles moved to another grid cells.

4 APPLICATION EXAMPLES

As it has been mentioned in the introduction, sorting algorithm has been included into particle simulation model. The simulation itself is performed on GPU as well, thus our approach is entirely computed on graphics processor. The choice of the particle model is quite arbitrary here, as we treat it mainly as a "wrapper" for the sorting algorithm. On the other hand it seems reasonable to pick up simulation that would give results understandable without deep insight into physical nature of the problem, and which wouldn't need too much
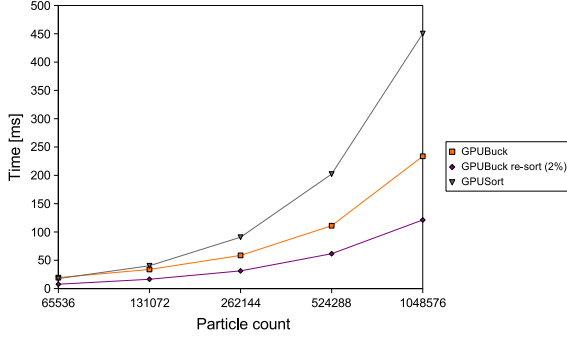
Particle Count	GPUBuck	2% re-sort	GPUSort
65536	19.34	7.68	17.67
131072	33.81	16.5	40.36
262144	58.61	31.27	90.75
524288	111.06	61.59	202.27
1048576	233.69	121.23	450.42

Table 3: GPUSort by Govindaraju compared to our bucket sorting implementation on a GeForce 6800 Ultra.

computing time. Having in mind three common particle models, namely Molecular Dynamics, Smoothed Particle Hydrodynamics and Dissipative Particle Dynamics, we have decided to rely on the latter one.

In this section we introduce briefly basic concepts of DPD model, the numerical method and the simulation conditions. Then we apply it to demonstrate the mixing of two immiscible fluids, driven by the Rayleigh-Taylor instabilities in a rectangular box, as well as to show the process of phase separation of two fluids.

4.1 Numerical model

In the DPD model [HK92] the discrete particles move about within the confines of a rectangular box with a height h and basis of L_x and L_y length. Periodic boundary conditions are imposed along the x- and y-direction, while reflecting boundary conditions are employed in the vertical z-direction. We have divided the box into two parts, with the upper (smaller) part of the box filled up with heavy fluid particle (H) and the lower part filled with lighter fluid particles (L). An external gravity field $\vec{G}$ pointing downwards is present. The particles are defined by the mass M_i, position r_i, and momentum p_i. We use classic two-body, short-ranged DPD force $\vec{F}_T = \vec{F}_C + \vec{F}_B + \vec{F}_D$. This type of force consists of conservative $\mathbf{F}_C$, dissipative $\mathbf{F}_D$ and Brownian (stochastic) $\mathbf{F}_B$ components. The value of $\mathbf{F}_C = \mathbf{F}_B = \mathbf{F}_D = 0$ for $r_{ij} > r_c$. Otherwise, we apply the following definitions:

$$
\begin{aligned}
\mathbf{F}_C &= \pi \omega_1(r_{ij})\mathbf{e}_{ij}, \\
\mathbf{F}_D &= \gamma M \omega_2(r_{ij})(\mathbf{e}_{ij} \cdot \mathbf{v}_{ij})\mathbf{e}_{ij}, \\
\mathbf{F}_B &= \frac{\sigma \Theta_{ij}}{\sqrt{\Delta t}}\omega_1(r_{ij})e_{ij}
\end{aligned}
$$

where: $\omega_1()$ and $\omega_2()$ - are the weight functions defined such that

$$
n_D \int_0^{r_c} \omega_m(r)d(r) = 1 \text{ for } m = 1,2.,
$$

r_{ij} – the distance between particles i and j, r_c – a cut-off radius, for which $\omega_1(r) = \omega_2(r) = 0$, n_D – an average particle density in D-dimensional system (D– dimension of the system), e_{ij} – a unit vector pointing from particle i to particle j, π – the scaling factor for the conservative part of collision operator, γ – the scaling factor for the dissipative force, σ – the scaling factor for the Brownian motion, Θ_{ij} – a random variable with a zero mean and actually normalised variance.

We assume that the normalised weight functions $\omega_1(r_{ij})$ and $\omega_2(r_{ij})$ are linear as it is in [ESZ97]). According to the fluctuation-dissipation theorem they are chosen such that $\omega_2(r_{ij}) = [\omega_1(r_{ij})]^2$ [CN96].

The temporal evolution of the particle ensemble obeys the Newtonian equations of motion. For integrating them we employ the "leap-frog" algorithm in time-steping for the particle positions r_i^n and the Adams-Bashforth scheme for the particle velocities v_i^n and momenta p_i^n. For the two-component fluid, where $k = g(i)$ and $l = g(j)$ denote the types of particle i and j (while $k,l \in H,L$), the equations of motion in 2-D space can be represented in the following discretized form.

$$
\begin{aligned}
\mathbf{p}_i^{n+\frac{1}{2}} &= \mathbf{p}_i^{n-\frac{1}{2}} + \sum_{i \neq j}\big[\pi_{kl}\omega_1(r_{ij}^n) \\
&\quad - \gamma_{kl}M_{kl}\omega_2(r_{ij}^n)\cdot(\mathbf{e}_{ij} \bullet \mathbf{v}_{ij}^n) \\
&\quad + \frac{\sigma_{kl}\Theta_{ij}}{\sqrt{\Delta t}}\omega_1(r_{ij}^n)\big]\mathbf{e}_{ij}\cdot\Delta t \\
\mathbf{r}_i^{n+1} &= \mathbf{r}_i^n + \frac{\mathbf{p}_i^{n+\frac{1}{2}}}{M}\Delta t \\
\mathbf{p}_i^n &= \frac{\mathbf{p}_i^{n+\frac{1}{2}} + \mathbf{p}_i^{n-\frac{1}{2}}}{2}
\end{aligned}
$$

Below we present snapshot from two DPD runs with 16384 particles. In the first run in figure 3 we demonstrate the process of two phase separation of particles in rectangular box. In the second simulation (figure 4) the gravity force acting downwards is added in the entire box. Starting from configuration, where heavier particles are placed in the top layer, we observe a development of Rayleigh-Taylor instability [Mik89].

4.2 Random number generator on the GPU

Dissipative Particle Dynamics method includes a Brownian component. To compute it on the GPU we used a random number generator which has been designed
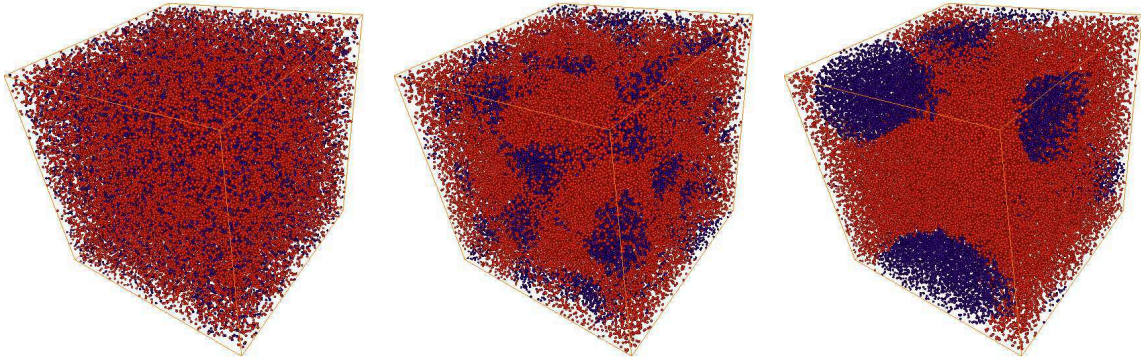
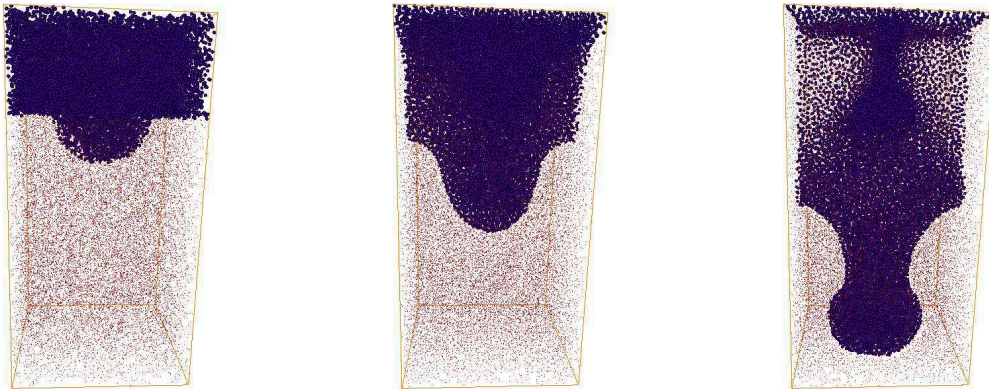Figure 3: Simulation of phase separation using DPD model with 65536 particles.

Figure 4: Simulation of Rayleigh–Taylor instability using DPD model, 65536.

to test floating-point behaviour of computer systems [Kar85]. The forumla is presented below:

$$x = 1.000005(r_i + \sqrt{3})^5$$
$$r_{i+1} = x - \lfloor x \rfloor$$

where the random-number r_{i+1} is computed from the previous value r_i. The initial value $r_0 = 0$. Its numerical properties are adequate to our needs and it can be easily implemented in a vertex or fragment shader.

5 CONCLUSIONS

We presented a novel way to implement bucket sorting on current graphics hardware. The results obtained are much faster than previous methods for at least some specific applications. Further we have shown the application of our algorithm to nearest neighbour search, which has been used in physical simulation. We used Dissipative Particle Dynamics to simulate fluids in real-time. We have also presented a random-number generated implementation on the GPU, which is required by the DPD.

Recently there has been another sorting algorithm published by Gress and Zachman [GZ06b] which outperforms GPUSort. Their implementation is a modified and optimised adaptive bitonic merge sort with a optimal complexity of $O(nlogn)$. We would like to compare our approach with these results in nearest future.

We also consider comparing OpenGL implementation with CUDA version on the new NVIDIA 8000 family.

REFERENCES

[AIY⁺04] Takashi Amada, Masataka Imura, Yoshi-hiro Yasumuro, Yoshitsugu Manabe, and Kunihiro Chihara. Particle-based fluid simulation on GPU. In *GP2 Workshop Proceedings*, August 2004.

[AMN⁺98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[BDH⁺06] Benjamin Bustos, Oliver Deussen, Stefan Hiller, , and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 196–199. Springer, 2006.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Al-*

gorithms. The MIT Press and McGraw-Hill Book Company, 1989.

[CN96]   Peter V. Coveney and Keir E. Novik. Computer simulations of domain growth and phase separation in two-dimensional binary immiscible fluids using dissipative particle dynamics. *Phys. Rev. E*, 54(5):5134–5141, Nov 1996.

[ESZ97]   P. Espanol, M. Serrano, and I. Zuniga. Coarse-graining of a fluid and its relation with dissipative particle dynamics and smoothed particle dynamics. *IJMPC*, 8(4):899–908, 1997.

[FS05]   Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[GRHM05]   Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, UNC, 2005.

[GZ06a]   Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, 25–29 April 2006.

[GZ06b]   Alexander Greß and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. Technical Report IfI-06-11, TU Clausthal, Computer Science Department, Clausthal-Zellerfeld, Germany, oct 2006.

[HCM06]   Kyle Hegeman, Nathan A. Carr, and Gavin S. P. Miller. Particle-based fluid simulation on the GPU. In *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 228–235. Springer, 2006.

[HK92]   P. J. Hoogerbrugge and J. M. V. A. Koelman. Simulating microscopic hydrodynamics phenomena with dissipative particle dynamics. *Europhysics Letters*, 19:155, 1992.

[HKK07]   Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, 2007.

[Kar85]   Richard Karpinski. PARANOIA: a floating-point benchmark. *Byte*, 10(2):223–235, 1985.

[KC05]   Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for GPU-based fluid simulation. *Proc. 18th Symposium on Simulation Technique*, pages 722–727, 2005.

[KSW04]   Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a GPU-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[LKO05]   Aaron Lefohn, Joe Kniss, and John Owens. Improved GPU sorting. In Matt Pharr, editor, *GPU Gems 2*, chapter 46, pages 733–746. Addison Wesley, March 2005.

[MCG03]   Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Mik89]   K. O. Mikaelian. Turbulent mixing generated by Rayleigh-Taylor and Richtmyer-Meshkov instabilities. *Physica D Nonlinear Phenomena*, 36:343–357, August 1989.

[OLG+05]   John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 24(3), 2005.

[PDC+03]   Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Proceedings of Graphics Hardware*, pages 41–50, 2003.

[Pur04]   Timothy John Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, March 2004.