A parallel SPH implementation on multi-core CPUs

Markus Ihmsen Nadir Akinci Markus Becker Matthias Teschner

University of Freiburg

Abstract

This paper presents a parallel framework for simulating fluids with the Smoothed Particle Hydrodynamics (SPH) method. For low computational costs per simulation step, efficient parallel neighborhood queries are proposed and compared. To further minimize the computing time for entire simulation sequences, strategies for maximizing the time step and the respective consequences for parallel implementations are investigated. The presented experiments illustrate that the parallel framework can efficiently compute large numbers of time steps for large scenarios. In the context of neighborhood queries, the paper presents optimizations for two efficient instances of uniform grids, i. e. spatial hashing and index sort. For implementations on parallel architectures with shared memory, the paper discusses techniques with improved cache-hit rate and reduced memory transfer. The performance of the parallel implementations of both optimized data structures is compared. The proposed solutions focus on systems with multiple CPUs. Benefits and challenges of potential GPU implementations are only briefly discussed.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism: —Animation

1. Introduction

Physically-based animation techniques are used to produce realistic visual effects for movies, advertisement and computer games. While the animation of fluids is becoming increasingly popular in this context, it is also one of the most computation-intensive problems. This paper focuses on the efficient simulation of fluids with Smoothed Particle Hydrodynamics (SPH) using multiple CPUs in parallel.

In SPH, the dynamics of a material is governed by the local influence of neighboring particles. In fluids, the set of neighboring particles dynamically changes over time. Therefore, the efficient querying and processing of particle neighbors is crucial for the performance of the simulation. The neighborhood problem for SPH fluids is similar to, e. g., collision detection or intersection tests in ray tracing. However, neighborhood queries in SPH are also characterized by unique properties that motivate our investigation of efficient acceleration data structures in this context.

For instance, adjacent cells in the employed spatial data structure have to be accessed efficiently. Further, an efficient data structure should employ the temporal coherence of a fluid between two subsequent simulation steps, but over larger time periods, it should preserve or restore spatial locality. As the acceleration structure has to be updated in each simulation step, query as well as construction times have to be optimized. Therefore, acceleration structures used in static applications, e. g. kd-trees or perfect hashing, might be too expensive to construct. Querying has to be efficient, as the set of neighboring particles has to be processed multiple times in an SPH algorithm for computing various attributes.

On average, a particle is interacting with 30 neighbors. Storing the neighbor set for fast reuse is, thus, a natural choice. However, for systems with a low-memory limit, this quickly limits the maximum complexity of the scene. In fact, recent SPH implementations on the GPU [HKK07b, ZSP08, GSSP10] do not store interacting particles, but recompute neighbor sets when needed. Consequently, the performance of these systems scales with the number of neighborhood queries in each simulation step. Therefore, either the gas equation [MCG03] or the Tait equation [Mon94, BT07] are employed, as these equations are fast to compute and the influencing particle sets are processed only twice per simulation step. However, the state equations suffer from visible compression artifacts, if the time step is not sufficiently small.

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

The predictive-corrective incompressible SPH method (PCISPH) [SP09] is more expensive to compute, but it can handle significantly larger time steps. In this method, incompressibility is enforced by iteratively predicting and correcting the density fluctuation. Due to the iteration scheme, the neighbor set is processed at least seven times per simulation step, which might limit the performance of current GPU implementations.

Our contribution. We present an efficient system for computing SPH fluid simulations on multi-core CPUs. Since the query and processing of particle pairs (neighbor search) is crucial for the performance, we focus on parallel spatial acceleration structures. We propose compact hashing and Z-index sort as optimizations of the commonly used spatial hashing and index sort schemes.

A low memory transfer is evident for a good performance of parallel systems. We employ techniques that lower the latency when accessing particles and their interacting neighbors. Temporal coherence is exploited in order to reduce grid construction times for both acceleration structures. We show that the performance of the proposed compact hashing is competitive with Z-index sort, while compact hashing is more appropriate for large scale fluid simulations in arbitrary domains.

In the context of SPH, the performance of compact hashing and Z-index sort is thoroughly analyzed. Both proposed schemes are compared with three existing variants of uniform grids, i. e. basic uniform grid, spatial hashing and index sort.

We further analyze the two types of SPH algorithms used in Computer Graphics, namely state equation based algorithms (SESPH) and the predictive-corrective SPH algorithm (PCISPH). We discuss the performance aspects of these algorithms. We finally show simulations with up to 12 million fluid particles using compact hashing and PCISPH.

2. Related work

In this work, we focus on an efficient fluid solver using the Smoothed Particle Hydrodynamcis method. While SPH is applied to model gas [SF95], hair [HMT01] and deformable objects [DC96,SSP07,BIT09], its main application in Computer Graphics is the simulation of liquids [MCG03]. In this field, research focuses on the simulation of various effects like interaction of miscible and immiscible fluids [MSKG05, SP08], phase transitions [KAG*05, SSP07], user-defined fluid animations [TKPR06] or the simulation and visualization of rivers [KW06].

Recent research is also concerned with performance aspects of SPH. Adams et al. [APKG07] use an adaptive sampling with varying particle radii. The number of particles inside a volume is reduced which significantly improves the performance for densely sampled fluids. However, in order to efficiently find interacting particles with different influence radii, a kd-tree is used which has to be rebuilt in every time step. According to the presented timings, the neighborhood search marks the performance bottleneck for this method.

There exist various approaches that address the implementation of SPH on highly parallel architectures, especially on the GPU. While SPH computations are easy to parallelize, the implementation of an efficient parallelized neighborhood search is not straightforward. In [AIY*04], e. g., the GPU is only used for computing the SPH update but not for the neighborhood query. Later, [HKK07b,ZSP08] presented SPH implementations that run entirely on the GPU. These methods map a three-dimensional uniform grid onto a two-dimensional texture. Particle indices are stored in RGBA channels. Thus, only four particles can be mapped into each grid cell. Due to this issue, a smaller than optimal cell size has to be chosen in order to avoid simulation artifacts.

Generally, even though a uniform grid is fast to construct and the costs of accessing an item are in $O(1)$, it suffers from a low cache-hit rate in the case of SPH fluid simulations. This is due to the fact that the fluid fills the simulation domain in a non-uniform way. Only a small part of the domain is filled, while the fluid also tends to cluster. Consequently, the grid is only sparsely filled and, hence, a significant amount of memory is unnecessarily allocated for empty cells. As stated in [HKK07b], this decreases the performance due to a higher memory transfer. In [HKK07a], an adaptive uniform grid is presented, where the memory consumption scales with the bounding box of the fluid volume.

Index sort is another approach to reduce the memory consumption and transfer of the uniform grid. Index sort first sorts the particles with respect to their cell indices c. Then, indices of the sorted array are stored in each cell. Each cell just stores one reference to the first particle with corresponding cell index. This idea has been described by Purcell et al. [PDC*03] for a fast photon mapping algorithm on the GPU. In [OD08], a similar idea is applied to a non-parallel SPH simulation. The paper shows that index sort outperforms the spatial hashing scheme for simulations with a low number of particles, i. e. 1300. Index sort is used in NVIDIA's CUDA based SPH fluid solver [Gre08] and also employed for fast parallel ray tracing of dynamic scenes [LD08, KS09]. In this work, we discuss important issues of the index-sort scheme in the context of a parallel SPH implementation on multi-core CPUs. Thereby, we propose a new variant named *Z-index sort* which includes SPH-specific enhancements.

In contrast to the uniform grid, the spatial hashing method can represent infinite domains. This method has been introduced for deformable solids [THM*03] and rigid bodies [GBF03]. For particle-based granular materials, spatial hashing is applied by [BYM05]. We propose a memory-efficient data structure for the spatial hashing method called *compact hashing* that allows for larger tables and faster queries.

In general, a hash function does not maintain spatial locality of data, which increases the memory transfer. In order to enforce spatial locality, we employ a space-filling Z-curve. A similar strategy is used by Warren and Salmon [WS95]. In this approach, a hashed octree is presented for parallel particle simulations. For the cosmological simulation code GADGET-2, particles are ordered in memory according to a Peano-Hilbert curve [Spr05]. The Peano-Hilbert curve preserves the locality of the data very well, but is relatively expensive to evaluate. Instead, we use the Z-curve which can be efficiently computed by bit-interleaving [PF01].

In addition to the uniform grid, spatial hashing and index sort, various other acceleration structures have been presented. One of the most popular techniques is the Verlet-list method [Ver67, Hie07]. In this method, a list of potential neighbors is stored for each particle. Potential neighbors have a distance which is lower or equal to a predefined range s, where s is significantly larger than the influence radius r. Thereby, the list of potential neighbors needs to be updated only if a particle has moved more than $s-r$. However, the memory transfer of this method scales with the ratio s/r, as all potential particle pairs are processed in each neighborhood query.

In [KW06], particles are sorted according to staggered grids, one for each dimension. Instead of querying spatially close cells in all dimensions at once, dimensions are processed one after another. As stated in [KW06], this method works well for a low number of particles. For higher resolutions, the performance is lower compared to, e. g., the octree used in [VCC98]. However, as reported in [PDC*03] and [HKK07b], hierarchical subdivision-schemes are not a good choice for fluids with uniform influence radius, since the costs of accessing an item are in $O(\log n)$, while for uniform grids they are in $O(1)$. This implies an higher memory transfer, which especially limits the performance of hierarchical data structures in parallel implementations.

The above-mentioned GADGET-2 simulation code is designed for massively parallel architectures with distributed memory. MPI (Message Passing Interface) is used for the parallelization. A parallel library for distributed memory systems is also presented by Sbalzarini et al. [SWB*06]. Using the library, continuum systems can be simulated with different particle-based methods. [FE08] use orthogonal recursive bisection for decomposing the simulation domain, in order to achieve load balanced parallel simulations of particle based fluids. This approach is designed for cluster systems. While these approaches focus on distributed memory systems, our approach addresses shared memory systems, where parallelization can be efficiently realized using OpenMP [Ope05].

3. SPH

The SPH method approximates a function $f(\mathbf{x}_i)$ as a smoothed function $\langle f(\mathbf{x}_i)\rangle$ using a finite set of sampling points $\mathbf{x}_j$ with mass m_j, density ρ_j, and a kernel function $W(\mathbf{x}_i-\mathbf{x}_j,h)$ with influence radius h. According to Gingold and Monaghan [GM77] and Lucy [Luc77], the original formulation of the smoothed function is

$$\langle f(\mathbf{x}_i)\rangle = \sum_j \frac{m_j}{\rho_j} f(\mathbf{x}_j) W(\mathbf{x}_i-\mathbf{x}_j,h). \tag{1}$$

The concept of SPH can be applied to animate different kinds of materials including fluids. The underlying continuum equations of the material are discretized using the SPH formulation. Thereby, objects are discretized into a finite set of sampling points, also called particles. The neighborhood of a particle i is defined by the particles j that are located within the support radius of i, i. e. $\|\mathbf{x}_i-\mathbf{x}_j \leq h\|$. Each particle represents the material properties of the object with respect to its current position. In each time step, these properties are updated according to the influence of neighboring particles. Therefore, the particle neighborhood has to be computed in each time step.

In the SPH method, derivatives are calculated by shifting the differential operator to the kernel function [Mon02]. Thereby, the computations are simplified. The most time consuming part of SPH simulations is the neighborhood query and the processing of neighbors, since the number of interacting particles (pairs) is significantly larger than the number of particles. In the following sections, we therefore focus on acceleration techniques and data structures for these methods.

4. Neighborhood search

The neighborhood search can be accelerated using spatial subdivision schemes. As reported in [HKK07b], the construction cost of a hierarchical grid is $O(n\log n)$ and the cost of accessing a leaf node is $O(\log n)$. In contrast, the construction cost of a uniform grid is $O(n)$, while any item can be accessed in $O(1)$. Therefore, uniform grid approaches are most efficient for SPH simulations with uniform support radius h.

4.1. Basic uniform grid

In the basic uniform grid approach, each particle i with position $\mathbf{x}=(x,y,z)$ is inserted into **one** spatial cell with coordinates (k,l,m). In order to determine the neighborhood of i, only particles that are in the same spatial cell or in one of the neighboring spatial cells within distance h need to be queried.

Obviously, the cell size d has an impact on the number of potential neighbors. The smaller the cell size, the smaller the number of potential pairs. However, with cell sizes smaller

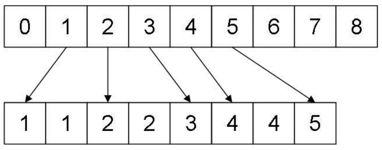

Figure 1: *Index sort data structure. The top row illustrates the uniform grid where the numbers refer to the corresponding cell indices. Each non-empty cell points to the first particle in the sorted particle array with corresponding cell index (bottom row).*

than h, the number of cells to query gets bigger. This might slow down the neighborhood query, due to a larger number of memory lookups. In the following, we assume a cell size of h. In this case, 27 cells have to be queried for each particle. We discuss the performance of different cell sizes in Sec. 6.

4.2. Index sort

Parallel construction. The parallel construction of the uniform grid is not straightforward since the insertion of particles into the grid might cause race conditions, i. e. two or more threads try to write to the same memory address concurrently.

As suggested in [PDC*03, KS09], these memory conflicts can be avoided by using sorted lists. The index sort method first sorts the particles in memory according to their cell index c. The cell index c of a position $\mathbf{x} = (x, y, z)$ is computed as

$$\begin{aligned} c &= k + l \cdot K + m \cdot K \cdot L \qquad (2)\\ \mathbf{c} &= (k, l, m) = \left(\left\lfloor \frac{x - x_{min}}{d} \right\rfloor, \left\lfloor \frac{y - y_{min}}{d} \right\rfloor, \left\lfloor \frac{z - z_{min}}{d} \right\rfloor \right) \end{aligned}$$

with d denoting the cell size. K and L may either denote the number of cells in x and y direction of the fluid's AABB or of the whole simulation domain.

In contrast to non-sorted uniform grids, a grid cell does no longer store references to all the particles in this cell. In fact, a cell with index c does only store one reference to the first particle in the sorted array with cell index c. We use a parallel reduction to insert the references into the grid [KS09]. Thereby, each bucket entry $B[k]$ in the sorted particle array B is tested against $B[k-1]$. Let c_k denote the cell index of the particle stored at $B[k]$. If c_k is different from c_{k-1}, a reference to $B[k]$ is stored in the spatial grid cell $G[c_k]$. This procedure can be performed in parallel since race conditions do not occur. The final data structure is illustrated in Fig. 1.

Index sort avoids expensive memory allocations while the memory consumption is constant. However, the whole spatial grid needs to be represented in order to find neighboring cells.

Parallel query. The neighborhood query for parallel architectures is straightforward. The sorted particle array is processed in parallel. For each particle i, all particles in the 27 neighboring cells are tested for interaction. Due to the sorting, particles that are in the same spatial cell are also close in memory. This improves the memory coherence (cache-hit rate) of the query. However, it depends on the indexing scheme if particles in neighboring cells are also close in memory. In the following section, we discuss important aspects when applying the index sort method on parallel CPU architectures.

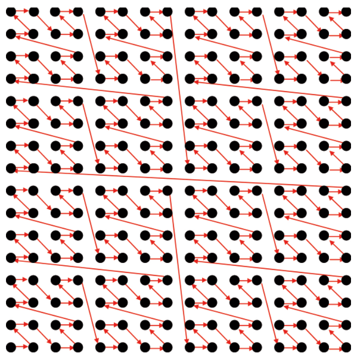

Figure 2: *The self-similar structure of the Z-curve in 2D is illustrated for a regular grid.*

4.3. Z-index sort

Indexing scheme. Current shared-memory parallel computers are built with a hierarchical memory system, where a very fast, but small cache memory supplies the processor with data and instructions. Each time new data is loaded from main memory, it is copied into the cache, while blocks of consecutive memory locations are transferred at a time. A new block may dispose other blocks of data, if the cache is full. Operations can be performed with almost no latency if the required values are available in the cache. Otherwise, there will be a delay. Thus, the performance of an algorithm is improved by reducing the amount of data transferred from main memory to the cache. Consequently, for parallel algorithms we have to enforce that threads do share as much data as possible without generating race conditions.

Even though it is not possible to directly control the cache, the program can be structured such that the memory transfer is reduced. The indexing scheme defined in (2) is not spatially compact, since it orders the cells according to their z position first. Thus, particles that are close in space are not necessarily close in memory. In order to further reduce the memory transfer, we suggest to employ a space-filling Z-curve for computing the cell indices.

Rather than sorting an n-dimensional space one dimension after another, a Z-curve orders the space by n-dimensional blocks of 2^n cells. This ordering preserves spatial locality very well due to the self-containing (recursive) block structure (see Fig. 2). Consequently, it leads to a high cache-hit rate while the indices can be computed fast by bit-interleaving [PF01]. In Sec. 6, we show that the Z-curve improves the cache-hit rate and, therefore, improves the performance for the query and processing of particle neighbors.

Sorting. The particles carry several physical attributes, e. g. velocity, position and pressure. When sorting the particles, these values have to be copied several times. Thus, the memory transfer might slow down the sorting significantly. In order to avoid sorting the particles array in every simulation step, we suggest to use a secondary data structure which stores a *key-value* pair, called handle. Each handle stores a reference to a particle (value) and its corresponding cell index (key). Due to the minimal memory usage of this structure, sorting the handles in each time step is much more efficient than sorting the particle array.

In order to yield high cache-hit rates, the particle array itself should still be reordered. However, we experienced that it is sufficient to reorder the particle array every 100th simulation step since the fluid simulation evolves slowly and coherently. The temporal coherence of the simulation data can be exploited further. According to the Courant-Friedrich-Levy (CFL) condition, a particle must not move more than half of its influence radius h in one time step. Thus, the average number of particles for which the cell index changes in a consecutive simulation step is low, i. e. 2%. Therefore, we propose to use *insertion sort* for reordering the handles. Insertion sort is very fast for almost sorted arrays. Usually, parallel *radix sort* is used for sorting [Gre08, OD08]. As we show in Sec. 6, the insertion sort outperforms our parallel radix sort implementation on CPUs even for large data sets.

Generally, index sort variants are considered to be the fastest spatial acceleration methods. However, according to [Gre08] there are two limitations. First, the memory consumption scales with the simulation domain. Second, for sorting, the whole particle array has to be reprocessed after computing the cell indices. In the following sections, we discuss an acceleration structure which is able to represent infinite domains.

4.4. Spatial hashing

In order to represent infinite domains with low memory consumptions, we employ the *spatial hashing* procedure introduced in [THM*03]. In this scheme, the effectively infinite domain is mapped to a finite list. The hash function that maps a position $\mathbf{x} = (x,y,z)$ to a hash table of size m has the following form:

$$c = \left[\left(\left\lfloor \frac{x}{d} \right\rfloor \cdot p_1 \right) \text{xor} \left(\left\lfloor \frac{y}{d} \right\rfloor \cdot p_2 \right) \text{xor} \left(\left\lfloor \frac{z}{d} \right\rfloor \cdot p_3 \right) \right] \% m \quad (3)$$

with p_1, p_2, p_3 being large prime numbers that are chosen similar to [THM*03] as 73856093, 19349663 and 83492791, respectively.

Unfortunately, different spatial cells can be mapped to the same hash cell (hash collision), slowing down the neighborhood query. The number of hash collisions can be reduced by increasing the size of the hash table. According to [THM*03], the hash table should be significantly larger than the number of primitives, in order to minimize the risk of hash collisions. Our experiments indicate that for SPH, a hash table size of two times the number of particles is appropriate.

In order to avoid frequent memory allocations, [THM*03] reserve memory for a certain number k of entries in all hash cells on initialization. Thereby, a table with m hash cells consumes $O(m \cdot k)$ memory. However, on average, the number of non-empty cells is only 12% of the total number of particles. For SPH fluids, the hash table is generally sparsely filled and a significant amount of memory is unnecessarily preallocated. Furthermore, for most of the hash cells, adjacent cells might be empty, which reduces the cache-hit rate for the insertion and query. In the following section, we present solutions to these problems.

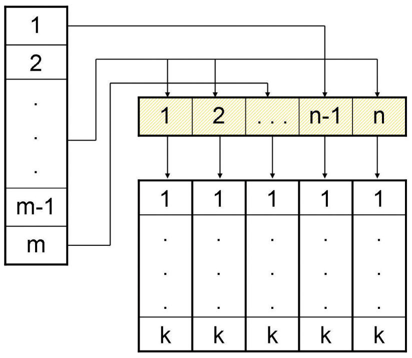

Figure 3: *Compact hashing. A handle array is allocated for the hash table of size m. The handles point to a compact list which stores the small number of n used cells (yellow shaded). For a used cell, memory for k entries is reserved. The memory consumption is thereby reduced to $O(n \cdot k + m)$. Note that the neighborhood query traverses only the n used cells.*

4.5. Compact hashing

In order to reduce the memory consumption, we propose to use a secondary data structure which stores a compact list of non-empty (used) cells. The hash cells do only store a handle to their corresponding used cell. Memory for a used cell is allocated if it contains particles and deallocated if the cell gets empty. Thus, this data structure consumes constant memory for the hash table storing the handles and additional memory for the list of used cells. In contrast to the basic uniform grid, the memory consumption scales with the number of particles and not with the simulation domain. The data structure is illustrated in Fig. 3.

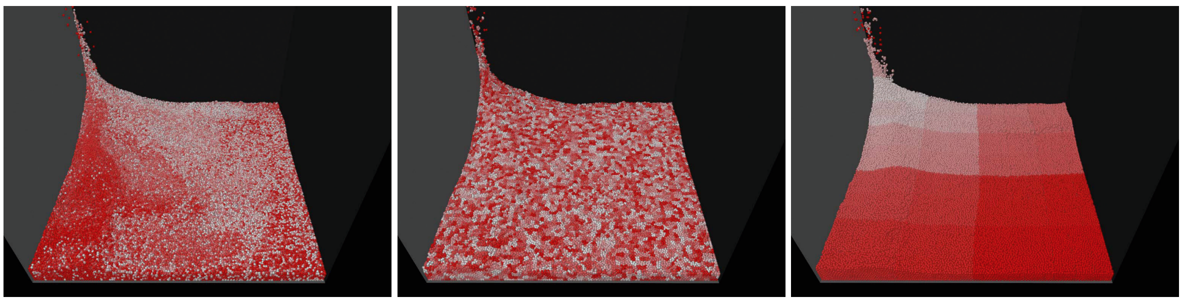

Figure 4: *Particles are colored according to their location in memory, where red is the last and white is the first position. Spatial locality is not maintained, if particles are not reordered (left). Since the hash function abolishes spatial locality, reordering according to the cell index (middle) is a bad choice to reduce the memory transfer. Spatial compactness is enforced using a Z-curve (right).*

The compact list of used cells already lowers the expected memory transfer. However, the hash function is not designed to maintain spatial locality, but rather abolishes it. Thus, compared to index sort, the required memory transfer for the query is still comparatively large. This again results in much larger query times, even if there are no hash collisions. In the following, we propose techniques, which significantly improve the performance of the construction and query for the spatial hashing method.

Parallel construction. Like for the non-sorted uniform grid, the particles cannot be inserted into the used cells in parallel. On the other hand, temporal coherence can be exploited more efficiently. Therefore, we do not reconstruct the compact list in each time step, but only account for the changes in each cell.

In a first loop, the spatial cell coordinates $\mathbf{c}_i = (k, l, m)$ of each particle i are computed. Only if $\mathbf{c}_i$ has changed, the new cell index of i is computed with (3) and the particle is added to a list of *moving particles*. In a second step, the moving particles are removed from their old cell and inserted into the new one. Note that the second step has to be performed serially, but with very low amortized costs since the number of moving particles is generally small, i. e. around 2% on average.

Parallel query. The neighborhood query is straightforward. The compact list of used cells can be processed in parallel without any race conditions. Nevertheless, the query is expected to be comparatively slow for the spatial hashing. This is due to hash collisions and an increased memory transfer, as consecutive used cells are close in memory, but not close in space. We now give solutions to overcome these limitations.

Hash collisions. For a used cell without hash collisions, all particles are in the same spatial cell and, hence, the potential neighbors are the same. However, if there is a hash collision in a used cell, the hash indices of the neighboring spatial cells have to be computed for each particle. This results in a significant computational overhead, particularly since we do not know if there is a hash collision in a cell or not. In order to keep the overhead low, we suggest to store a hash-collision flag in each used cell. Therefore, the hash indices have to be computed only once for cells without hash collisions.

The memory consumption of the proposed data structure scales with the number of particles and not with the hash (handle) table size. Therefore, the hash table can always be set to a size which enforces a very low number of hash collisions. For SPH, this number is usually below 2%. Storing a collision flag significantly speeds up the query since the performance is almost not influenced by the low number of hash collisions.

Spatial locality. Hash functions are designed to map data from a large domain to a small domain. Thereby, the data is scattered and spatial locality is usually not preserved. The list of used cells is, hence, not ordered according to space which means that consecutive cells are not spatially close (see Fig. 4). Thus, an increased memory transfer is expected for the query since already cached data might not be reused.

In order to reduce the memory transfer, we again sort the particles according to a Z-curve every 100th simulation step. Sorting is performed similar as described in Sec. 4.3. Instead of sorting the particle array with all attributes, we sort a handle structure. Each handle consists of cell index and reference to a particle. Then, all other attributes of the particle are sorted accordingly.

Note that when sorting the particle array, the pointers in the used cell become invalid. Therefore, the compact used-cell list has to be rebuilt. Used cells are created each time a particle is added to an empty hash cell. Thus, if we simply insert the sorted particles serially into the compact hash table, the order of the used cells is consistent with the Z-curve order of the particles. We employ a similar parallel reduction scheme as in Sec. 4.2, in order to parallelize this step. The computational overhead of reconstructing the compact list of

Algorithm 1: SESPH

```
foreach particle i do
    compute density ;
    compute pressure (4) or (5);
foreach particle i do
    compute all forces;
    integrate;
```

used cells is negligible since reordering every 100th steps is sufficient.

In this section, we described a spatial hashing scheme optimized for SPH simulations on parallel architectures. By allocating memory only for non-empty hash cells, the overall memory consumption is reduced. Furthermore, for the neighborhood query, only the small percentage of used cells is traversed, which significantly improves the performance. Finally, for shared memory systems, the memory transfer is minimized by reordering particles and the compact list according to a space-filling curve.

5. Fluid update

In Computer Graphics, two different algorithms are generally used for updating SPH fluids, namely SESPH [MCG03, BT07] and PCISPH [SP09]. The neighborhood search discussed in Sec. 4 builds the base for these algorithms since physical attributes are updated according to the influence of neighboring particles. This is illustrated in Alg. 1 and Alg. 2 where we marked the steps that process the neighbor set with red and bold letters. In this section, we discuss performance and accuracy issues of both SPH algorithms.

5.1. SEPSPH

The SESPH algorithm loops only two times over all particles in each simulation step (see Alg. 1). Particles and their neighbors are processed in two subsequent loops, in order to update the physical attributes.

For SESPH, the pressure can be computed by using an equation of state, namely the Tait-equation [BT07] or the gas equation [MCG03]. The Tait-equation relates the pressure p with the density ρ polynomially as

$$p = k\left(\left(\frac{\rho}{\rho_0}\right)^7 - 1\right) \tag{4}$$

where ρ_0 denotes the rest density and k is a stiffness parameter that governs the relative density fluctuation $\rho - \rho_0$. While for (4), the pressure grows polynomially with the compression of the fluid, for the gas equation

$$p = k(\rho - \rho_0) \tag{5}$$

it just grows linearly. Consequently, (4) results in larger pressures than (5) which restricts the time step. Furthermore, as

Algorithm 2: PCISPH

```
foreach particle i do
    compute forces $\mathbf{F}_i^{\nu,st,ext}(t)$;
    set pressure to 0;
    set pressure force to $(0,0,0)^T$;
$k = 0$ ;
while ($max(\rho^*_{err_i}) > \eta$ or $k < 3$) do
    foreach particle i do
        predict velocity ;
        predict position ;
    foreach particle i do
        update distances to neighbors;
        predict density variation;
        update pressure ;
    foreach particle i do
        compute pressure force;
    $k += 1$;
foreach particle i do
    integrate;
```

reported in [BT07], k should not be set too small in order to avoid compression artifacts. In order to get plausible simulation results with SEPSH, small time steps are required.

5.2. PCISPH

In contrast to SESPH, the PCISPH algorithm [SP09] does not use an equation of state for computing the pressure. This method predicts and corrects the density fluctuations in an iterative manner. Thereby, the density error is propagated until the compression is resolved. However, in every simulation step, at least three iterations are required in order to limit temporal fluctuations (see Alg. 2). Thereby, the neighbor set is processed 7 times in total. Thus, the PCISPH method is comparatively expensive to compute.

5.3. Performance comparison

For SPH simulations, various authors [HKK07b, Gre08, GSSP10] give the number of simulation updates per lab second and refer to this as frames per second (fps). In our opinion, this number gives only small insights about the overall performance as long as the time step is not given. In order to assess the performance of our system, we do not only state the simulation updates per second, but also the time step. In the following, ups denotes the number of times the simulation is updated (neighbor search + fluid update) in one lab second.

Accordingly, the PCISPH might be more efficient than the SESPH method, despite the computational overhead per fluid update. The reason is that the PCISPH method can handle significantly larger time steps. [SP09] reports up to 150

	recompute neighbors		store neighbors	
particles	SESPH	PCISPH	SESPH	PCISPH
130K	47.1	190.1	44.4	105.4
1700K	662.1	2649.5	572.3	1378.7

Table 1: *Performance comparison of storing the neighbors and recompute them on-the-fly. Here, SESPH and PCISPH times (in milliseconds) include the neighbor query, but not the construction time of the grid. Simulations were performed on an Intel Xeon 7460 using all 24 CPUs with 2.66GHz.*

method	construction	query	total
basic uniform grid	25.7 (27.5)	38.1 (105.6)	**63.8** (133.1)
index sort [Gre08]	35.8 (38.2)	29.1 (29.9)	**64.9** (77.3)
Z-index sort	16.5 (20.5)	26.6 (29.7)	**43.1** (50.2)
spatial hashing	41.9 (44.1)	86.0 (89.9)	**127.9** (134.0)
compact hashing	8.2 (9.4)	32.1 (55.2)	**40.3** (64.6)

Table 2: *Performance analysis of different spatial acceleration methods with and without (in brackets) reordering of particles. Timings are given in milliseconds for CBD 130K and include storing of pairs.*

times larger time steps in comparison to SESPH using the Tait-equation [BT07].

Also, SESPH using the gas equation requires much smaller time steps than the PCISPH in order to achieve plausible simulation results. For all of our simulations, the time step for PCISPH could be at least set 25 times higher than for SESPH with (5). However, SESPH updates the simulation only 2.5 times faster than PCISPH. Accordingly, the PCISPH outperforms both SESPH algorithms at least by a factor of ten.

Storing or recomputing neighbors. For some platforms, dynamic memory management is expensive or challenging. In such cases, recomputing the particle neighbors on-the-fly can be efficient. Consequently, state-of-the art SPH implementations on the GPU do not store particle pairs, but recompute them when needed. In contrast, for our multi-core CPU systems, we experienced that performing the neighbor search only once per simulation step and query the pairs from memory is more efficient.

In Table 1, we analyze the performance difference for the fluid update, when querying pairs from memory and recomputing the neighborhood on-the-fly. When pairs are stored, neighbors are computed in a first loop and then written to and read from memory. In Table 1, these times are included for *store neighbors*. On our system, the overhead of writing and reading the data to memory pays off. While the benefit is rather low for SESPH, it is significant for PCISPH. This is due to the fact, that PCISPH performs seven neighbor queries, if pairs are not stored. Accordingly, for PCISPH, the neighbor query becomes the bottleneck if neighbors need to be recomputed, as for the GPU based systems presented in [HKK07b, ZSP08, Gre08, GSSP10].

6. Results

In this section, we evaluate the performance of the investigated data structures. We start with a detailed performance analysis of the five uniform grid variants. By comparing the construction and query times, we show that the proposed Z-index sort and the compact hashing are most efficient. Next, we discuss the performance influence of the cell size and the scaling of our system for a varying number of threads. Finally, we give complete analyses for some example scenes, where we also compare the performance of PCISPH and SESPH. In the following, PCISPH and SEPSH times are for reading neighbors from memory. Time required for neighbor search is listed separately.

Our test system has the following specifications:

- CPU: 4x 64-Bit Intel Six Core 7460@2.66GHz, 16 MB shared L3 cache, 64MB Snoop Filter that manages cached data to reduce data traffic
- Memory: 128GB RAM, Bus Speed 1066MHz

6.1. Performance analysis

We use **one** example scene for analyzing the performance of the presented spatial acceleration methods. For this scene, we have chosen a corner breaking dam with 130K particles (CBD 130K) since it is a typical test scene in the field of computational fluid dynamics. We averaged the performance over 20K time steps (see Table 2).

Basic uniform grid. The construction of the basic uniform grid is not parallelized due to race conditions. The query step loops over all particles in parallel and computes the neighbors. Thereby, we do not have to traverse the whole, sparsely filled grid. However, if particles are not reordered, the memory transfer is high, which has a significant impact on the performance.

Index sort. In comparison to the basic uniform grid, the query of the index sort is much faster since the query processes the particles in the order of their cell indices. Consequently, the cache-hit rate is high, which improves the performance significantly. However, the construction time is comparatively large due to a slow sorting time. Our parallel radix sort algorithm sorts the 130K key-value pairs (handles) in 25ms which is slow compared to the much faster multi-core sort implementation described in [CNL*08]. Note that for this method, particles are reordered according to their cell index which is computed with (2).

Z-index sort. For Z-index sort, the handles are sorted using insertion sort instead of radix sort. Due to the small

method	compact hashing	PCISPH (SESPH)	ups
no reorder	64.6	171.8 (38.4)	4.2 (9.6)
reorder	40.3	78.6 (17.6)	8.3 (16.7)

Table 3: *Performance improvement when particles are reordered every* 100 *step. ups denotes simulation updates (neighbor search + fluid update) per lab second. Timings are in milliseconds for CBD 130K.*

changes from one time step to the next, only a small number of particles move to another spatial cell. Thus, the handles are only slightly disordered from one time step to the next. Since insertion sort performs very well on almost sorted lists, the sorting time is significantly reduced to $6ms$. Furthermore, in contrast to index sort, the cell indices are computed on a Z-curve and the particles are reordered accordingly. By mapping spatial locality onto memory, the cache-hit rate is increased and, hence, the construction and query time for Z-index sort is further reduced.

Spatial hashing. We have implemented the spatial hashing method described in [THM*03]. This method performs the worst due to hash collisions and a high memory transfer invoked by the hash function. Note that even when reordering the particles according to a Z-curve, the improvement is marginal since we traverse the whole hash table and spatially close cells are not close in memory. However, if we loop over the particles and not the whole grid, spatial compactness can be exploited. In this case, the query time reduces to $50ms$ when particles are reordered (this number is not given in Table 2).

Compact hashing. The proposed compact hashing exploits temporal coherence in the construction step. Thereby, the insertion of particles into the grid is five times faster than for spatial hashing. Furthermore, the query is nearly three times faster due to the reduced memory transfer invoked by the compact list of used-cells and the hash collision flag.

Reordering. Note that we reorder the particles every 100th step according to a Z-curve in all methods except index sort. By reordering the particles according to a Z-curve, spatially close particles reside close in memory. Thus, particles that are close in memory are very likely spatial neighbors. Since in SPH, most computations are interpolations requiring informations of neighbors, the Z-curve increases the cache-hit rate for the neighborhood query and for the fluid update. Reordering significantly reduces the memory transfer and thereby improves the performance (see Table 3).

We summarize that compact hashing is as efficient as Z-index sort. Both methods outperform spatial hashing and the basic uniform grid. For compact hashing, the memory consumption scales with the number of particles, while for Z-index sort it scales with the domain. Therefore, for very large or unrestricted domains, compact hashing should be preferred. On the other hand, if influencing pairs can not be stored, Z-index sort should be used since the query time is lower.

cell size	pairs	tested pairs	ratio	query
h	3.6M	25M	6.9	26.6
$0.5h$	3.6M	15.5M	4.3	39.6

Table 4: *Influence of the cell size on the ratio of tested pairs to influencing pairs and the query time in millisecond. Test scene CBD 130K.*

CPUs	compact hashing	Z-index sort	PCISPH	SESPH
1	404.1	391.3	791.1	189.5
24	40.3	43.1	78.6	17.6

Table 5: *Parallel scaling of neighbor search and simulation update. For 24 CPUs a speed up of* 10 *is achieved. Timings are in milliseconds for CBD 130K.*

6.2. Cell size

The size of the spatial cells influences the number of potential pairs that have to be tested for interaction. Generally, smaller cell sizes approximate the shape of the influence radius h better which is a sphere in 3D. If the cell size d is chosen as h, only 27 cells need to be tested for interaction. If $d < h$, the number of potential cells grows, but less particles have to be tested for interaction. However, if more cells are queried, the memory transfer increases. We observed that the increased memory transfer is more expensive than testing more potential neighbors for interaction (see Table 4). Thus, the optimal cell size is the influence radius.

6.3. Parallel scaling

Optimally, the speed up from parallelization would be linear. However, the optimal scaling can not be expected due to the parallelization overhead for synchronization and communication between different threads. According to Amdahl's law [Amd67], even a small portion of the problem which cannot be parallelized will limit the possible speed up. For example, if the sequential portion is 10%, the maximum speed up is 10.

Rewriting algorithms, in order to circumvent data dependencies is fundamental to increase the possible speed up. Furthermore, the performance of an algorithm is either memory or CPU bound. While for CPU-bound algorithms the performance is easily increased by using an additional or faster CPU, the bottleneck for memory-bound problems is the bandwidth. Therefore, only strategies that are reducing the memory transfer are improving the efficiency.

The presented strategies and techniques employed in Z-index sort and the compact hashing as well as the reordering

scene	# particles	compact hashing [ms]	PCISPH (SESPH) [ms]	$\Delta t[s]$	ups
Glass	75K (20K)	30.8	29.8 (7.3)	0.0025 (0.0001)	16.5 (26.2)
CBD 130K	130K	40.3	78.6 (17.6)	0.0006 (1.8e-5)	8.3 (16.7)
CBD large	1.7M	427.8	1130.0 (271.6)	0.0002 (5.7e-6)	0.6 (1.4)
River	12M (5M)	5193.6	11941.8	0.0005	0.06

Table 6: *Average performance in milliseconds for different scenes. Compact hashing is used in all scenes. For the glass and river scene, we use boundary particles (numbers are given in brackets). Fluid update times, time step and simulation updates per lab second (ups) are for PCISPH and in brackets for SESPH (similar simulation result).*

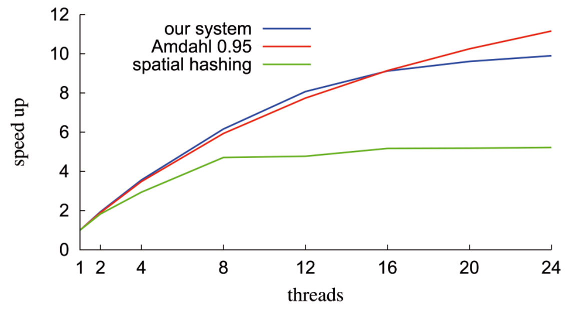

Figure 5: *Total speed up of our system for fluid update and neighbor search using compact hashing. The scaling is in good agreement with Amdahl's law. For comparison, the scaling of spatial hashing is given.*

of particles lower the memory transfer and, thus, improve the performance of the system significantly. The scaling and even the flattening of the proposed system is in very good agreement with Amdahl's law for a problem that is parallelized to 95% (see Fig. 5 and Table 5). In contrast, the scaling of spatial hashing is much worse.

6.4. Scaling with particles

Finally, we show that our system scales linearly with the number of particles (see Table 6). We therefore set up different small-scale and large-scale simulations. In the *glass scene*, a glass is filled with 75K particles (see Fig. 7). For this scene, a plausible simulation result is achieved by PCISPH for a time step of 0.0025. Thus, ten *real world* seconds were simulated in four minutes. With SESPH we computed a similar result with a 25 times smaller time step in 63 minutes using (5).

Due to the large memory capacity of CPU architectures, our system shows good performance for large-scale simulations with millions of particles. For those setups, querying the particles multiple times has a significant impact on the performance. On average, for the *CBD large scene*, 341 million pairs are queried for interaction per simulation step and 2.3 billion pairs for the *river scene* (see Fig. 6 and Fig. 8).

Our system updates an SESPH simulation of 130K particle with 17 ups. This performance is similar to the fastest GPU implementation [GSSP10]. However, we present simulations with up to 12 million particles using the more efficient PCISPH method, while [GSSP10] shows simulations with up to 250K particles using SESPH.

7. Conclusion

We presented a parallel CPU-based framework for SPH fluid simulations. Important aspects which are critical for the performance of such a system are discussed. For acceleration structures based on uniform grids, the construction and query times are reduced by lowering the memory transfer. This is achieved by mapping spatial locality onto memory, using compact data structures and exploiting temporal coherence. Furthermore, we showed how the spatial hashing can be optimized for a parallel SPH framework. We thoroughly analyzed the performance aspects of the five presented uniform grid methods and give detailed scaling analyses. Additionally, we investigated the performance of different SPH algorithm, i. e. PCISPH and SESPH.

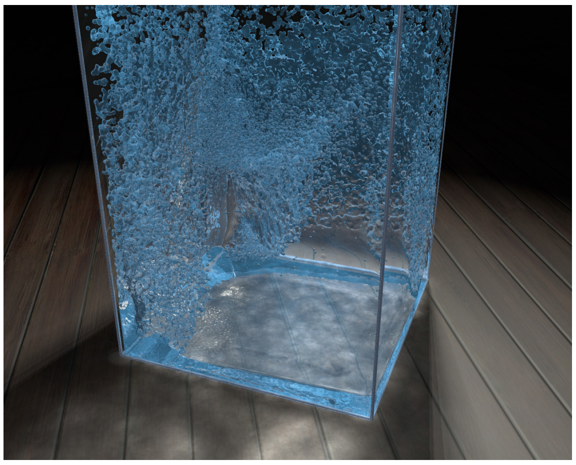

Figure 6: *CBD large scene. A corner breaking dam with 1.7 million particles simulated with PCISPH.*

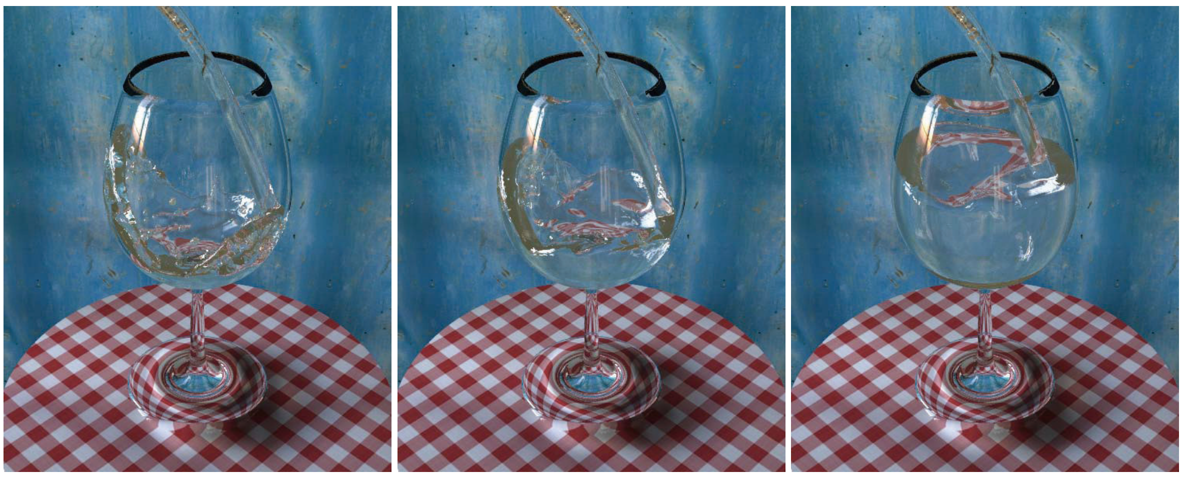

Figure 7: *Glass scene. The 'wine' is sampled with up to 75K particles, while the glass is sampled with 20K boundary particles. Interaction is computed with [BTT09].*

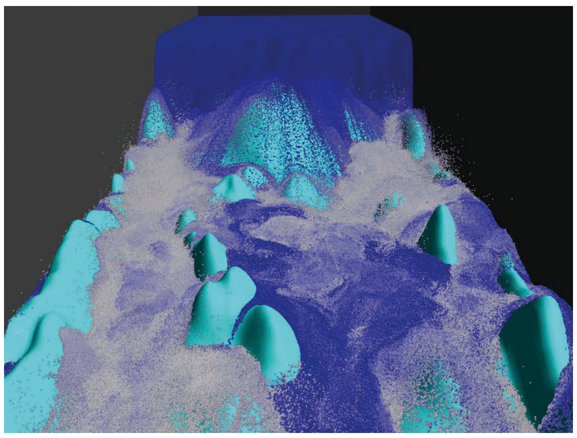

Figure 8: *River scene. The fluid consists of 12.1 million particles and the terrain is sampled with more than 5 million particles. The particles are coded according to acceleration, where white is high and blue is low.*

Acknowledgements

This project is supported by the German Research Foundation (DFG) under contract number TE 632/1-1. We thank the reviewers for their helpful comments.

References

[AIY*04] AMADA T., IMURA M., YASUMORO Y., MANABE Y., CHIHARA K.: Particle-based fluid simulation on GPU.

[Amd67] AMDAHL G.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings* (1967), vol. 30, pp. 483–485.

[APKG07] ADAMS B., PAULY M., KEISER R., GUIBAS L.: Adaptively sampled particle fluids. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers* (New York, NY, USA, 2007), ACM Press, p. 48.

[BIT09] BECKER M., IHMSEN M., TESCHNER M.: Corotated SPH for deformable solids. *Eurographics Workshop on Natural Phenomena* (2009), 27–34.

[BT07] BECKER M., TESCHNER M.: Weakly compressible SPH for free surface flows. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, 2007), Eurographics Association, pp. 209–217.

[BTT09] BECKER M., TESSENDORF H., TESCHNER M.: Direct forcing for lagrangian rigid-fluid coupling. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (2009), 493–503.

[BYM05] BELL N., YU Y., MUCHA P. J.: Particle-based simulation of granular materials. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2005), ACM Press, pp. 77–86.

[CNL*08] CHHUGANI J., NGUYEN A. D., LEE V. W., MACY W., HAGOG M., KUANG CHEN Y., BARANSI A., DUBEY P.: Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. In *34th Intel Conference on Very Large Data Bases* (2008), pp. 1313–1324.

[DC96] DESBRUN M., CANI M.-P.: Smoothed Particles: A new paradigm for animating highly deformable bodies. In *Eurographics Workshop on Computer Animation and Simulation (EGCAS)* (1996), Springer-Verlag, pp. 61–76. Published under the name Marie-Paule Gascuel.

[FE08] FLEISSNER F., EBERHARD P.: Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. *International Journal for Numerical Methods in Engineering 74* (2008), 531–553.

[GBF03] GUENDELMAN E., BRIDSON R., FEDKIW R.: Nonconvex rigid bodies with stacking. *ACM Trans. Graph. 22*, 3 (2003), 871–878.

[GM77] GINGOLD R., MONAGHAN J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society 181* (1977), 375–398.

[Gre08] GREEN S.: Particle-based Fluid Simulation. http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf, 2008.

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive SPH Simulation and Rendering on the GPU. In *SCA '10: Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2010).

[Hie07] HIEBER S.: *Particle-Methods for Flow-Structure Interactions*. PhD thesis, Swiss Federal Institute of Technology, 2007.

[HKK07a] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Sliced data structure for particle-based simulations on gpus. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (New York, NY, USA, 2007), ACM, pp. 55–62.

[HKK07b] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Smoothed Particle Hydrodynamics on GPUs. In *Proc. of Computer Graphics International* (2007), pp. 63–70.

[HMT01] HADAP S., MAGNENAT-THALMANN N.: Modeling Dynamic Hair as a Continuum. *Computer Graphics Forum 20*, 3 (2001), 329–338.

[KAG*05] KEISER R., ADAMS B., GASSER D., BAZZI P., DUTRÉ P., GROSS M.: A Unified Lagrangian Approach to Solid-Fluid Animation. In *Proceedings of the Eurographics Symposium on Point-Based Graphics* (2005), pp. 125–134.

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics* (New York, NY, USA, 2009), ACM, pp. 23–28.

[KW06] KIPFER P., WESTERMANN R.: Realistic and interactive simulation of rivers. *Proceedings of the 2006 conference on Graphics interface* (2006), 41–48.

[LD08] LAGAE A., DUTRÉ P.: Compact, fast and robust grids for ray tracing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks* (New York, NY, USA, 2008), ACM, pp. 1–1.

[Luc77] LUCY L.: A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal 82* (1977), 1013–1024.

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 154–159.

[Mon94] MONAGHAN J.: Simulating free surface flows with SPH. *Journal of Computational Physics 110*, 2 (1994), 399–406.

[Mon02] MONAGHAN J.: SPH compressible turbulence. *Monthly Notices of the Royal Astronomical Society 335*, 3 (2002), 843–852.

[MSKG05] MÜLLER M., SOLENTHALER B., KEISER R., GROSS M.: Particle-based fluid-fluid interaction. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2005), ACM, pp. 237–244.

[OD08] ONDERIK J., DURIKOVIC R.: Efficient Neighbor Search for Particle-based Fluids. *Journal of the Applied Mathematics, Statistics and Informatics (JAMSI) 4*, 1 (2008), 29–43.

[Ope05] OPENMP ARCHITECTURE REVIEW BOARD: *OpenMP Application Program Interface, Version 2.5*, May 2005.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 41–50.

[PF01] PASCUCCI V., FRANK R. J.: Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 2001), ACM, pp. 2–2.

[SF95] STAM J., FIUME E.: Depicting fire and other gaseous phenomena using diffusion processes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), ACM Press, pp. 129–136.

[SP08] SOLENTHALER B., PAJAROLA R.: Density Contrast SPH Interfaces. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2008).

[SP09] SOLENTHALER B., PAJAROLA R.: Predictive-Corrective Incompressible SPH. In *SIGGRAPH '09: ACM SIGGRAPH 2009 Papers* (2009). to appear.

[Spr05] SPRINGEL V.: The cosmological simulation code GADGET-2. *Mon. Not. R. Astron. Soc. 364* (2005), 1105–1134.

[SSP07] SOLENTHALER B., SCHLÄFLI J., PAJAROLA R.: A unified particle model for fluid-solid interactions. *Computer Animation and Virtual Worlds 18*, 1 (2007), 69–82.

[SWB*06] SBALZARINI I., WALTHER J., BERGDORF M., HIEBER S., KOSALIS E., KOUMOUTSAKOS P.: PPM - A highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comp. Phys. 215*, 2 (2006), 566–588.

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANETS D., GROSS M.: Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proc. of Vision, Modeling, Visualization (VMV)* (2003), pp. 47–54.

[TKPR06] THÜREY N., KEISER R., PAULY M., RÜDE U.: Detail-preserving fluid control. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2006), Eurographics Association, pp. 7–13.

[VCC98] VERMURI B., CAO Y., CHEN L.: Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum 17*, 2 (1998), 121–134.

[Ver67] VERLET L.: Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys. Rev. 159*, 1 (1967), 98–103.

[WS95] WARREN M., SALMON J.: A portable parallel particle program. *Computer Physics Communications 87*, 1-2 (1995), 266–290.

[ZSP08] ZHANG Y., SOLENTHALER B., PAJAROLA R.: Adaptive Sampling and Rendering of Fluids on the GPU. In *Proceedings Symposium on Point-Based Graphics* (2008), pp. 137–146.