Volume xx (200y), Number z, pp. 1–11

Compressed Neighbor Lists for SPH

Stefan Band[1], Christoph Gissler[1,2] and Matthias Teschner[1]

[1]University of Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany
[2]FIFTY2 Technology GmbH, Tullastraße 80, 79108 Freiburg im Breisgau, Germany
{bands,gisslerc,teschner}@informatik.uni-freiburg.de

Abstract

We propose a novel compression scheme to store neighbor lists for iterative solvers that employ Smoothed Particle Hydrodynamics (SPH). The compression scheme is inspired by Stream VByte, but uses a non-linear mapping from data to data bytes, yielding memory savings of up to 87 %. It is part of a novel variant of the Cell-Linked-List (CLL) concept that is inspired by compact hashing with an improved processing of the cell-particle relations. We show that the resulting neighbor search outperforms compact hashing in terms of speed and memory consumption. Divergence-Free SPH (DFSPH) scenarios with up to 1.3 billion SPH particles can be processed on a 24-core PC using 172 GB of memory. Scenes with more than 7 billion SPH particles can be processed in an MPI environment with 112 cores and 880 GB of RAM. The neighbor search is also useful for interactive applications. A DFSPH simulation step for up to 0.2 million particles can be computed in less than 40 ms on a 12-core PC.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Physical simulation;*** *Massively parallel and high-performance simulations;*

1. Introduction

Approaches for the SPH neighbor search rarely discuss the storage of the computed neighbor lists. This is due to the fact that the storage of neighbor lists had not necessarily been required in state-equation solvers with very few iterations over particle neighbors per simulation step. Also, storing neighbors can constitute a significant memory overhead which is particularly prohibitive for GPU implementations.

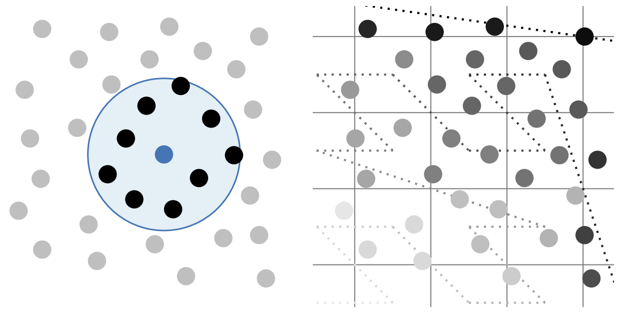
Current SPH approaches, however, employ an increasing number of neighbor iterations. State-of-the-art incompressibility solvers such as PCISPH [SP09], IISPH [ICS*14], DFSPH [BK17] and PBF [MM13] are iterative. Boundary handling can be iterative as proposed in, e.g., Pressure Boundaries [BGI*18] or for strong solid-fluid coupling [GPB*19]. And there exist a growing number of implicit formulations for, e.g., viscous [PT17, WKBB18], elastic [PGBT18] or ferromagnetic materials [HHM19] that employ iterative solvers. In such iterative solvers, storing and re-using neighbor lists is a natural choice.

This paper focuses on the storage of compressed neighbor lists in iterative SPH solvers. We propose a novel compressed representation of neighbor lists that is inspired by Stream VByte [LKR17]. Further, the compression scheme is embedded into a novel variant of the Cell-Linked-List (CLL) approach, e.g. [HGE74, DCGGM11, IABT11]. We show that the proposed neighbor search and query outperform state-of-the-art methods such as compact hashing [IABT11] in terms of memory consumption and computing time. Using our neighbor processing, fluid scenes with up to 200 million SPH particles can be computed on a single PC with 32 GB memory.

The proposed compression scheme works on a particle list that is sorted with respect to the index of a space-filling curve computed for a virtual uniform grid. Figure 1 illustrates the setting and Fig. 2 indicates the sorted particle list where particles in the same space cell are adjacent. This array is queried to find the neighbor list for each particle. Figure 3 shows the same array with highlighted particles that constitute the neighbor list of an exemplary particle. The fact that all neighbor lists are represented by a smaller number of clusters in the sorted array is the starting point for the proposed compression.

1.1. Our contribution

- We propose to store compressed neighbor lists in SPH. We show that this concept enables fluid scenarios with 1.3 billion particles on a single PC and more than 7 billion particles on six connected PCs.
- We propose a novel scheme for the compressed storage of



(a) SPH approximations at a particle (blue) require neighboring particles (black).

(b) Virtual space cells. The dotted line illustrates the order of the cells.

Figure 1: In order to accelerate the neighbor search within the blue-shaded area in (a), ordered virtual cells shown in (b) are considered. Each particle stores the index of its cell illustrated by the particle shading. Particles in the same cell have the same cell index.

Figure 2: Particles sorted with respect to the cell index. Particles that share the same cell are adjacent in this list.

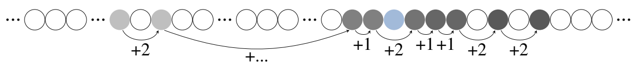


Figure 3: A typical set of neighbors computed on the sorted particles. The numbers indicate index differences in the sorted array. We propose a novel compression scheme to store the index differences.

integer-valued index differences. In contrast to, e.g. Stream VByte [LKR17], small values are directly encoded in the control bits without using any data byte. On the other hand, values of intermediate size (encoded with two or three data bytes in Stream VByte) require four bytes in our approach. We show that the proposed non-linear mapping of values to data bytes outperforms Stream VByte in SPH scenarios due to the clustered representation of neighbors in the sorted particle list.

- Our compressed representation is integrated in a novel CLL approach. We propose a novel compact and memory-efficient representation of the cell-particle relation in combination with an adapted neighbor query using the BigMin-LitMax algorithm [TH81]. In contrast to [DCGG13], infinite domains can be handled by not representing empty cells. In contrast to compact hashing [IABT11], only one particle reference is stored per used cell. We show that our CLL approach is faster than compact hashing. In particular, 0.2 million SPH particles can be interactively simulated on a 12-core PC with our approach.
- We present various analyses. We show performance comparisons to compact hashing and memory comparisons to Stream VByte. We show that differences in computing times are negligible when using compressed instead of uncompressed neighbor lists. We further show that compressed neighbor lists are particularly advantageous for larger kernel supports with an increasing number of neighbors where memory savings of up to 87 % can be obtained. Finally, we illustrate the parallelizability and scalability of the neighbor search and storage by presenting scenarios on a 12- and 24-core PC and in an MPI environment with 112 cores.

2. Related Work

2.1. Compression

Our work focuses on integer-valued compression techniques. In the sorted array of a CLL approach, SPH particles are identified by indices represented with 32-bit unsigned integers. Also, index differences are integer values. There exist a large variety of suitable compression methods and recent surveys can be found, e.g., in [ZZL*15] and [LB15].

We consider methods that encode a block of integer values with control and data bytes. In binary packing a fixed number of data bits is used for each value within the same block of, e.g., 128 integers [NAM10]. The decreasing efficiency of this fixed pattern in case of outliers in a block has been addressed, e.g., by the patched frame of reference (PFOR) [ZHNB06]. Recent PFOR implementations identify outliers in a sequence and process them separately, e.g., NewPFD [YDS09], OptPFD [YDS09], FastPFOR [LB15].

As the SPH neighbor lists are represented in a smaller number of clusters in the sorted array of our CLL approach, we prefer compression schemes that handle small index differences within a cluster and few large index differences between clusters with an adaptive number of data bits. E.g., VByte [TH72] is a popular choice to compress sequences of integers with variable data patterns, e.g. [PKL15]. VByte is byte-oriented, but its implementation involves a branch and potential branch mispredictions may impair the performance.

VarIntGB [Dea09] alleviates the performance problems of VByte. It is also well suited to make use of modern hardware acceleration (SIMD) [SGR*11, LB15, ZZL*15]. However, in VarIntGB there is a strong data dependency in the decoding procedure, as the location of the next control byte depends on the current control byte. This increases the risk that the processor remains underutilized, as it cannot load and start processing upcoming control bytes.

Stream VByte [LKR17] overcomes this data dependency by storing the control bytes in a stream that is separate from the data bytes. In their experiments, [LKR17] reported Stream VByte to be the fastest of the above-mentioned algorithms. Therefore, we base our compression scheme on Stream VByte. In contrast to Stream VByte, however, we propose a non-linear mapping from integer values to the number of data bytes. In particular, the values one and two are represented in the control byte without the usage of any data byte. We show that this variant outperforms Stream VByte in the specific context of the CLL-based neighbor search in SPH simulations.

2.2. Neighbor Search

SPH requires information of neighboring particles to compute the particle interactions. Using a naive approach, in which each particle is a potential neighbor of all other particles, is unfeasible for a large number of particles N due to a computational complexity of $\mathcal{O}(N^2)$ and is also unnecessary since particle pairs that are farther than the SPH support length have a kernel function value of zero and can thus be discarded.

To accelerate the neighborhood query, hierarchical tree data structures that can be built in $\mathcal{O}(N \log N)$ and queried in $\mathcal{O}(\log N)$ have been applied, e.g. [WS95, KAG*05, HCM06, APKG07], however, mostly in astrophysical problems with long-range interactions [Spr05, GR11, HBMW11]. In contrast, researchers in computer graphics seem to prefer uniform grids, e.g. [Gon07, Gre10, PH10, DCGGM11, DCVB*13, IOS*14, WMRR17, WSG*18] that can be built in $\mathcal{O}(N)$ and queried in $\mathcal{O}(1)$.

In addition to uniform grids, Verlet-Lists (VL) [Ver67, VBC08, WMRR17] have been proposed to alleviate the performance issue of re-computing the neighbor lists in each simulation step. As the neighbor lists are created from a larger support, they can be retained for several steps. Due to memory restrictions on low memory systems like GPUs, using the VL approach might not always be an option. In this case, a CLL approach is preferred, e.g. [HGE74, HKK07, GSSP10, DCGG13].

CLL approaches require the construction of a data structure to link a cell of the underlying uniform grid with a list of particles. In GPU implementations, e.g. [Gre10, DCVB*13, GEF15, MRSD15, WRR18], all grid cells are typically represented, including empty ones. For sparsely filled domains, where the number of grid cells is much larger than the number of particles, this approach is not memory-efficient. In order to represent only non-empty cells, [Gre10, IABT11, TLTM18] propose to use an approach based on spatial hashing. However, hash tables are designed to scatter data according to spatially close cells in order to minimize hash collisions. This leads to low cache-hit rates for the insertion and query. Further, race conditions while inserting, e.g., due to hash collisions, have to be handled appropriately.

In contrast to previous works, we address the neighbor search problem by a novel compact and memory-efficient representation of the CLL in combination with an adapted neighbor query. Thereby, we avoid the issues associated with hashing while infinite domains can be handled with low memory consumption.

3. Method

This section describes the concept of our novel CLL variant. We start with the proposed compression scheme for neighbor lists in Section 3.1. Afterwards, the embedding of the compressed neighbor lists into the novel CLL variant is explained in Section 3.2. Implementation details are given in the following Section 4.

3.1. Compression

We want to compress neighbor lists that are represented in the sorted array of our CLL approach (see Fig. 3). If n_i is a particle index in the sorted array and a neighbor list consists of N particles, we are interested in a lossless compression of N strictly increasing non-negative integers $n_1, n_2, \ldots, n_N$. As motivated in Section 2, our compression scheme is inspired by Stream VByte [LKR17]. Stream VByte encodes the number of non-zero data bytes required to represent an integer in a 2-bit binary mask (see Table 1). Four of these binary masks build one control byte. As the number N of integers is known, the first $\lceil 2N/8 \rceil$ bytes in the output sequence are used to store the control bytes. The control bytes are followed by the data bytes

	Stream VByte		Our approach	
mask	integer	data bytes	integer	data bytes
`0b00`	$[0, 2^8)$	1	0	-
`0b01`	$[2^8, 2^{16})$	2	1	-
`0b10`	$[2^{16}, 2^{24})$	3	$[2, 2^8)$	1
`0b11`	$[2^{24}, 2^{32})$	4	$[2^8, 2^{32})$	4

Table 1: Control bits and encoded value ranges for Stream VByte [LKR17] and our compression approach.

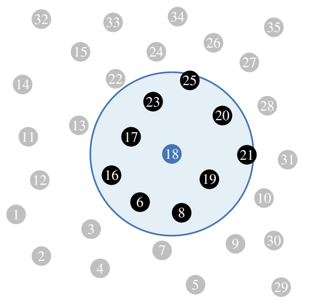
which are made up of the non-zero bytes of the input integers. For efficiency, integers are encoded and decoded in blocks of four by utilizing SIMD hardware instructions. To encode such a block, 5 to 17 bytes are required.

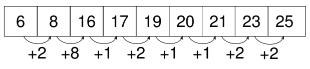
To minimize the range of the considered data, deltas or gaps with $\Delta n_i = n_i - n_{i-1}$ are useful. In our context, many deltas are small and thus well compressible [PKL15]. To recover the original data from the deltas, the first value n_1 of a sequence has to be stored together with the deltas. Then, a prefix sum $n_i = n_{i-1} + \Delta n_i$ is used to reconstruct the rest of a sequence [LF80]. The value of each recovered integer, however, can only be calculated after the preceding value is known. To alleviate this problem, deltas could be computed on a four-by-four basis, i.e., $\Delta n_i = n_i - n_{i-4}$ as proposed in [LB15]. Another approach is to compute the deltas according to the minimal value, i.e., the first value in our application: $\Delta_i = n_i - n_1$. This approach is commonly referred to as frame of reference (FOR) [GRS98]. Although both approaches are faster than the original differential coding, they tend to generate larger deltas and inferior compression ratios.

In our specific application of SPH neighbor lists, the deltas are always greater or equal to one. Subtracting one from each delta makes them even smaller, but still non-negative, i.e., $\Delta n_i = n_i - n_{i-1} - 1 \geq 0$. As discussed in Section 5.1 and illustrated in Fig. 7 for an exemplary scenario, many deltas are either zero or one. In consequence, we propose to improve the compression rate of Stream VByte by introducing a non-linear mapping from deltas to data bytes. Instead of using 1, 2, 3 or 4 data bytes, we encode the values 0 or 1 with control bits only, and use 1 or 4 data bytes for all other values (see Table 1). No data bytes are required for the values 0 and 1. Values in the interval $[2, 2^8)$ are represented with one data byte. Values larger or equal to 2^8 are represented with four data bytes.

The compression algorithm for a list of neighbors works as follows. The first neighbor n_1 is stored uncompressed. Then, deltas are computed with $\Delta n_i = n_i - n_{i-1} - 1$ for all other neighbors in the list. From the deltas, we gather the control bits as specified in Table 1 and store the final control byte sequence. For deltas not equal to zero or one, we encode Δn_i with one or four data bytes and store the final data byte sequence.

For decoding, these steps are reversed. We read a control byte, extract the control bits, read the specified number of data bytes and compute a prefix sum using the first uncompressed value n_1 to finally restore the original values $n_2, \ldots, n_N$. Figure 4 illustrates an exemplary setting.

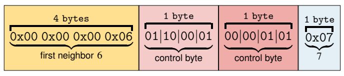

(a) Particle configuration. The numbers indicate the particle index in the sorted array.



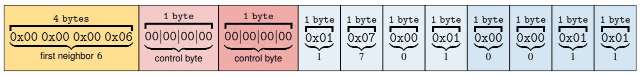
(b) Neighbor set of particle 18. The numbers at the arrows correspond to index differences of the particle indices. Storing the uncompressed neighbor list would require $9 \times 4\,\mathrm{B} = 36\,\mathrm{B}$.

6	1	7	0	1	0	0	1	1

(c) The delta encoded neighbor list considering unique particle indices. The first neighbor (yellow) is not delta encoded.



(d)
ing this list requires 4 B for the first neighbor (yellow), $2 \times 1\,\mathrm{B}$ for the control bytes (red) and 1 B for the data segment (blue), i.e., a total of 7 B.



(e) With Stream VByte [LKR17], the neighbor list can be compressed to $4\,\mathrm{B} + 2 \times 1\,\mathrm{B} + 8 \times 1\,\mathrm{B} = 14\,\mathrm{B}$.

Figure 4: Compression concept for a neighbor list and comparison to Stream VByte.

3.2. Neighbor Search

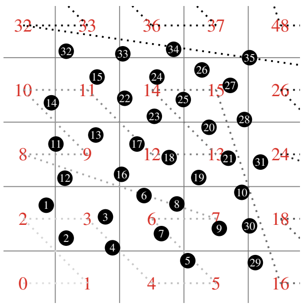
Our compression scheme bases on a novel CLL variant for the neighbor search. In particular, the CLL query has to compute neighbors with sorted indices to guarantee non-negative deltas for the compression. Further, our CLL variant is motivated by memory efficiency in support of the compressed neighbor lists.

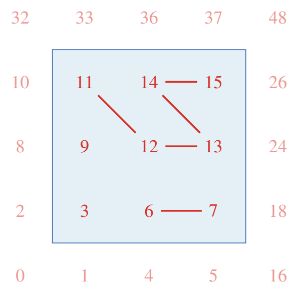
We consider a 3D grid that consists of equally sized cube cells, e.g. [KS09, Gre10]. A particle with position $r_i = (x, y, z)$ is mapped to a 3D coordinate of a grid cell with

$$(k, l, m) = \left\lfloor \frac{r_i - r^{\min}}{\hbar} \right\rfloor \tag{1}$$

where we subtract the minimum position $r^{\min} = (\min x, \min y, \min z)$ to ensure that all grid coordinates (k, l, m) are positive. The grid is only implicitly represented by the edge length of a cell, i.e., the SPH kernel support $\hbar$ in our application. Each 3D coordinate (k, l, m) is then mapped to a unique 1D cell index $c \equiv c(k, l, m)$, e.g [PDC*03, KS09, IABT11] (see Fig. 5a).

In order to improve spatial locality and in turn reduce memory transfer by improving the cache-hit rate, we compute the cell index based on Morton codes [Mor66]. If k, l and m are represented by B bits, i.e., $k = k_1 \cdots k_B$, $l = l_1 \cdots l_B$ and $m = 1 \cdots m_B$, the cell index c is computed by bit interleaving, i.e., $c(k, l, m) = k_1 l_1 m_1 \cdots k_B l_B m_B$ [PF01, LSY01]. After that, the particle list is sorted according to this index. Although parallel radix sort is often used for sorting, e.g. [OD08, Gre10, GSSP10], we employ a parallel stable merge sort [MRR12] as it outperforms our radix sort implementation even

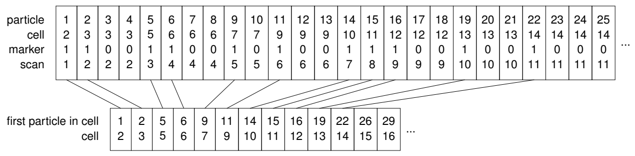

(a) Each particle is mapped to a virtual grid cell with a unique cell index c. The particles are ordered in memory according to c.



(b) The neighbor search can be simplified by range queries. The red lines represent ranges of contiguous cell indices that can be computed with the BigMin-LitMax algorithm [TH81]. In this example, the query rectangle $[3, 15]$ is split into the four sub-ranges 3, $[6, 7]$, 9 and $[11, 15]$.

particle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...
cell	2	3	3	3	5	6	6	6	7	7	9	9	9	10	11	12	12	12	13	13	13	14	14	14	14	

(c) Particles and their corresponding cells after sorting with respect to the cell index for the setting shown in (a).



particle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	...
cell	2	3	3	3	5	6	6	6	7	7	9	9	9	10	11	12	12	12	13	13	13	14	14	14	14	
marker	1	1	0	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0	0	1	0	0	0	
scan	1	2	2	2	3	4	4	4	5	5	6	6	6	7	8	9	9	9	10	10	10	11	11	11	11	

first particle in cell	1	2	5	6	9	11	14	15	16	19	22	26	29	...
cell	2	3	5	6	7	9	10	11	12	13	14	15	16	

(d) Compact representation of the cell-particle relationships in (c). Due to the ordering of the particles, it is sufficient to only store a reference to the first particle inside each non-empty cell. For instance, cell 9 contains the particles $[11, 14)$ whereas cell 12 contains the particles $[16, 19)$. Querying all particles in the cell range $[11, 15]$ would yield the particle range $[15, 29)$. Marker and scan values are just computed in order to generate the compact cell list. They are not stored.

Figure 5: CLL approach with the proposed cell-particle relations and the BigMin-LitMax algorithm to estimate relevant cells for a neighbor query.

for a large number of particles. In the sorted array, particles that belong to the same grid cell are stored contiguously in memory (see Fig. 5c).

In our novel CLL variant, we propose to build a compact list of all non-empty cells from the sorted particle list. For each non-empty grid cell, we store its cell index and a reference to its first particle in the sorted particle list. This is in contrast to compact hashing [IABT11], where references to all particles in a cell are stored. It is also in contrast to [DCGG13], where all empty and non-empty cells are represented and store references to their first and last particle in the sorted particle list. We can infer the number of particles inside a cell by computing the difference between the particle reference of the current to the particle reference of the next grid cell in the compact list. To create the compact cell array, first, a marker is computed from the sorted particle list (see Fig. 5d). A particle is marked with one if its cell value differs from the preceding particle in the sorted list, or with zero otherwise. A parallel scan (prefix sum) over the markers provides the offsets for the compacted cell array. Note that the marker and scan value are only computed, but not stored. Finally, the particle index is written to the compact cell array at the scan location if the particle's marker value is one.

For the neighborhood query, particles inside a given bounding box must be found. We define a bounding box as a range between a minimum and a maximum cell index and adopt an idea of [DCGG13]. As the grid cells follow the order of the underlying curve, consecutive cells can be queried together (see Fig. 5b). However, not all cells in a given range contain potential particle neighbors. In order to maximize query performance, the range is split into smaller sub-ranges. To compute the sub-ranges, we propose to use the BigMin-LitMax algorithm [TH81, OM84]. Finally, to compute the lower and upper particle index bound for a sub-range, we perform a ternary search on the cell array with logarithmic complexity in the number of non-empty grid cells. Due to an ordered processing of the sub-ranges, the indices of the estimated neighbors in a set are ordered which is a prerequisite for our proposed compressed neighbor lists. This also results in an optimized caching and hardware pre-fetching behavior for the SPH computations.

4. Implementation

The implementation of our CLL approach is summarized in Algorithm 1. First, the particle array is sorted with respect to the cell index. Then, marker and scan values as indicated in Fig. 5d are computed on the particle array to enable the generation of the compact cell array. After that, the BigMin-LitMax algorithm is performed for all cells in the compact cell array. The particles in the resulting sub-ranges of cells are potential neighbors. Found neighbors are compressed and stored per particle. All loops are well suited for parallelization as there are no data dependencies.

Most relevant indices and references are simply computed or directly looked up in the two arrays. The minimum and maximum cell indices computed by the BigMin-LitMax algorithm, however, have to be searched in the compact cell array. Here, we employ a ternary search algorithm with a fallback to linear search [SGL09]. The search interval is recursively cut in three parts until the remaining search interval contains less than 16 elements. For the remaining elements, we fall back to a linear search algorithm. Linear search has a worse theoretical run-time complexity compared to ternary search. In practice, however, it can be more efficient for searching small intervals as it is suitable for modern SIMD-hardware. Thus, it can be implemented in a conditional-free way while testing multiple elements at once. For further performance improvements, we cache the last 1024 results of the search algorithm. If a previously processed value is found in the cache, it can be re-used without search.

5. Results

SPH setting: We have integrated our novel CLL variant and the compressed neighbor representation into a DFSPH framework [BK17, BGPT18]. If not stated otherwise, we use the Wendland C^6 kernel [Wen95] with a support $\hbar$ of two times the particle spacing h

```
1:  For each particle $i$
2:      Compute grid cell coordinate $(k_i, l_i, m_i)$ with Eq. (1)
3:      Compute and store grid cell index $c_i \equiv c_i(k_i, l_i, m_i)$
4:  Sort the particles based on their cell index $c_i$. Now, the particles
    configuration is as illustrated in Figs. 5a and 5c
5:  Generate and fill the compact cell array. The size of the compact
    cell array is given by the maximum scan value as depicted in
    Fig. 5d
6:  For each cell in the compact cell array
7:      For each particle $i$ in the current cell
8:          Create an empty local neighbor set $\mathcal{N}_i$
9:      Determine the query box range for the neighbor search and
        use the BigMin-LitMax algorithm to split the query box into
        smaller sub-ranges as indicated in Fig. 5b
10:     For each sub-range
11:         Use a ternary search on the compact cell array on the
            sub-range boundaries to get a particle range. This range
            is made of a lower and upper particle index bound. All
            particles within this range are potential neighbors
12:         For each particle $j$ in the computed particle range
13:             For each particle $i$ in the current cell
14:                 If particles $i$ and $j$ interact, i.e., $||r_i - r_j|| < \hbar$
15:                     add $j$ to $i$'s local neighbor set $\mathcal{N}_i$
16:     For each particle $i$ in the current cell
17:         Compress the local neighbor set $\mathcal{N}_i$ with our algorithm
            specified in Section 3.1 and store it to global memory
```

Algorithm 1: Query and compression of particle neighbors.

for all SPH interpolations. We follow [BL99, Gan15] and use linear-exact gradient estimates for computing the pressure acceleration and divergence of the velocity.

Simulation features: In free-surface scenarios, we apply a drag force to the fluid-air interface as described in [GBP*17] and model surface tension as proposed in [AAT13]. Viscosity is modeled as proposed by [MFZ97]. Solid boundaries are represented with a single particle layer of non-uniform size [AIA*12]. Pressure forces at solid boundaries are computed with extrapolated pressure according to [BGPT18].

Hardware and software: The presented scenarios have been computed on three different systems. 1. One workstation with 12 cores and 32 GB of RAM (2.6 GHz Intel Xeon E5-2690). 2. One workstation with 2 × 12 cores and 256 GB of RAM (2×2.7 GHz Intel Xeon E5-2697). 3. Six workstations with 112 cores and 880 GB of RAM (two 2×12-core 2.7 GHz Intel Xeon E5-2697 and four 2×8-core 3.1 GHz Intel Xeon E5-2687W). The six workstations are connected via Gigabit Ethernet. All computations are fully parallelized with Intel Threading Building Blocks [Phe08]. For the six connected workstations, an MPI environment is realized. Like [TSG14], we use Orthogonal Recursive Bisection (ORB) [Fox88] for spatial domain decomposition to distribute the particles to the computation nodes. The load balancing is done with a proportional integral controller [FE08, OLTG*16]. The ray-traced images are rendered with PreonLab [FIF19]. The accompanying video is generated with 50 frames per second.

Scenarios: We use a dam break (Fig. 6) and a wind tunnel (Fig. 10) for our analyses. For the dam break, we follow [KFV*05] and simulate a fluid of size $1\,\mathrm{m} \times 0.55\,\mathrm{m} \times 1.228\,\mathrm{m}$ inside a box-shaped domain of size $1\,\mathrm{m} \times 1\,\mathrm{m} \times 3.22\,\mathrm{m}$. Furthermore, there is an obstacle of size $0.403\,\mathrm{m} \times 0.161\,\mathrm{m} \times 0.161\,\mathrm{m}$ in the scene. If not stated otherwise, we use a particle spacing of 4 mm, resulting in 10.5 million fluid and 0.8 million boundary particles. The rest density of the fluid is $1000\,\mathrm{kg\,m^{-3}}$. Furthermore, we use a fixed time step size of 0.1 ms and a fixed number of two divergence-free and two density-invariant iterations of the DFSPH solver.

The wind tunnel scene consists of 1272 million fluid particles and 30.1 million boundary particles with a spacing of $h = 4\,\mathrm{mm}$ and a support length of $2h$. The size of the simulation domain is $4\,\mathrm{m} \times 4\,\mathrm{m} \times 10\,\mathrm{m}$. We use a fixed time step of 0.5 ms and set the density of the fluid to $1.2041\,\mathrm{kg\,m^{-3}}$ which corresponds to the density of dry air at $20\,^{\circ}\mathrm{C}$. The inlet flow velocity of the fluid is $4\,\mathrm{m\,s^{-1}}$.

5.1. Space Filling Curves

The space filling curve is an important degree-of-freedom in our CLL approach that affects the performance of the index computation and neighbor list creation. It also influences the compression rate of the stored neighbor lists. Therefore, we compare XYZ (also used in [PDC*03, DCGG13]), Morton (also used in [IABT11]) and Hilbert curves (also used in [GRLS18]) with respect to performance and compressibility using the dam break scenario. All considered curves are discussed, e.g. in [Bad12] .

In a first experiment, we measure the wall-clock time to compute the curve index, i.e., the cell index for the particles (Lines 1 to 3 in Algorithm 1). Furthermore, we measure the time required to search and store the (compressed) neighbor lists for the particles (Lines 6 to 17 in Algorithm 1). This is an indicator of the spatial locality that is obtained by a space filling curve. As spatial locality also influences all SPH computations, we also measure the total computation time required per simulation step including neighbor search and storage, pressure computation and particle advection. The results of our timing experiments are summarized in Table 2.

The measurements in Table 2 indicate that the curves have varying performance in the cell index computation, the neighbor list creation and the SPH computations. E.g., the index computation of a Hilbert curve is rather expensive and the neighbor list creation is comparatively slow. Interestingly, the SPH computations seem to benefit from the Hilbert curve which results in an improved overall performance compared to the XYZ curve. The Morton curve resulted in the best performance for creating the neighbor lists and also in the best overall performance.

In a second experiment, we measure how well the tested space filling curves are suited for the compression of the neighbor lists. After finding all particle neighbors and storing them in ascending sorted order, we compute deltas $\Delta n_i = n_i - n_{i-1} - 1$ for all particles and

	average computation time per time step [s]		
curve	cell index	neighbor list creation	total
Hilbert	0.017	0.404	2.846
Morton	0.004	0.287	2.747
XYZ	0.004	0.331	3.077

Table 2: Performance comparison of three space filling curves for the dam break scenario using a 12-core PC.

their neighbors (see Fig. 7). All tested curves result in similar distributions. For the Morton [Mor66] and Hilbert curve [Hil91], however, approximately 93 % of the differences lie in the range $[0, 256)$. As these values are higher than the 86 % for the XYZ curve, the Morton and Hilbert curve should lead to improved compression rates. As the Morton curve has an improved overall performance compared to the Hilbert curve, we propose to sort space cells, i.e., particles, according to the Morton curve.

5.2. Compression Scheme

We compare the compression ratio of our scheme to existing variants within our CLL approach. We also measure the time to search and store all particle neighbors and we measure the total time to compute a simulation step. We particularly compare to the uncompressed case to illustrate the memory saving, but also the computing overhead for encoding and decoding of the neighbor lists. The measurements in Table 3 show that the memory requirement for a neighbor index can be reduced from 4 B in the uncompressed case to 0.851 B using our compression scheme. This corresponds to an average memory saving of 79 %. In particular, our variant requires less memory than Stream VByte. The encoding and decoding overhead for all computations within a simulation step is approximately 1 % compared to the uncompressed case. Figure 8 illustrates that the average size of a neighbor set can be reduced from 120 B in the uncompressed case to 25 B using the proposed scheme.

Our compression scheme yields the best compression rate. This is due to the proposed handling of small differences between two consecutive neighbor indices. On the other hand, the cost for storing larger differences is higher with our compression scheme compared to Stream VByte. As large differences are only an exceptional case in our SPH fluid setting, our proposed compression method outperforms Stream VByte. Although fewer bytes are read from memory, our experiments indicate that decoding the neighbor list is associated with a small performance overhead.

5.3. CLL Approach

In Table 4, we compare the performance and memory consumption of our CLL variant with compact hashing as proposed in [IABT11]. Our CLL approach requires significantly less memory than compact hashing. Building and querying the respective data structures to estimate and store the neighbor lists is also faster with the proposed compacted cell array and search algorithm.

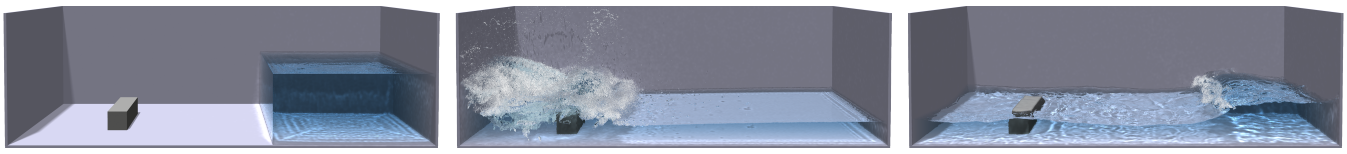


Figure 6: The dam break scenario at $t = 0$ s (left), $t = 1.5$ s (middle) and $t = 3$ s (right).

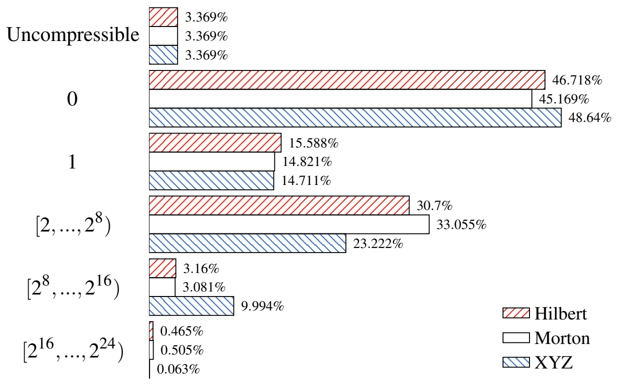


Figure 7: Particle neighbor delta distribution for the dam break scenario. The first neighbor is not compressible since it is required to restore the original values.

	average per time step		
	computation time [s]		memory [B]
method	neighbor list creation	total	per neighbor
Uncompressed	0.274	2.721	4.000
Stream VByte	0.285	2.730	1.445
Our approach	0.287	2.745	0.851

Table 3: Comparison of different compression methods for the breaking dam scenario using a 12-core PC.

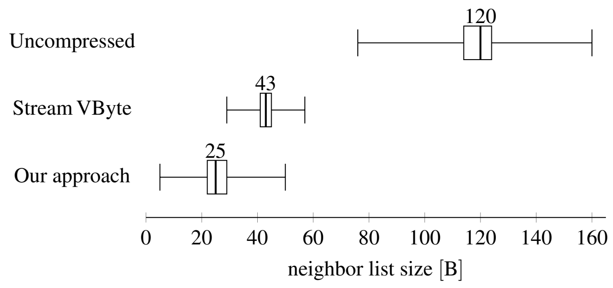


Figure 8: Distribution of the memory requirements for the neighbor lists of the breaking dam scenario. The central mark indicates the median. The left and right box edges indicate the lower and upper quartiles. The whiskers extend to the most extreme data points which are no more than 1.5 times the interquartile range from the box.

	average per time step			
	computation time [s]			memory [MB]
method	build	n. l. creation	total	total
Compact hashing	0.043	0.374	2.868	136.3
Our approach	0.004	0.287	2.747	15.1

Table 4: Comparison of two CLL variants for the dam break scenario using a 12-core PC. The memory requirement is significantly reduced with our approach due to the compact cell array.

	average per time step			
		computation time [s]		memory [B]
support	neighbors per particle	n. list creation	total	per neighbor
2.0 h	29.88	0.287	2.746	0.851
2.5 h	67.50	0.558	5.856	0.726
3.0 h	104.64	0.740	8.366	0.662
3.5 h	183.92	1.114	14.335	0.609
4.0 h	255.52	1.485	19.690	0.579
4.5 h	396.69	2.189	30.092	0.552
5.0 h	483.93	2.909	36.980	0.536

Table 5: Larger supports result in larger numbers of neighbors and in improved compression ratios. Numbers are given for the dam break scenario using a 12-core PC.

5.4. Scaling with the Number of Neighbors

The support length $\hbar = sh$ with $s \in \mathbb{R}^+$ governs the number of particle neighbors. E.g., increasing the support length from $2h$ to $3h$ results in an increase from approximately 30 to 105 neighbors per particle. While larger supports are interesting in terms of simulation accuracy, their usage is typically limited by the significant increase in memory consumption. Our compressed neighbor lists alleviate this issue.

Table 5 shows performance and memory measurements for varying supports. It can be seen that the compression ratio gets better for a growing support. These measurement illustrate that our compressed neighbor lists and the entire CLL approach are particularly useful for SPH simulations with larger supports. This is also emphasized in Fig. 9 where memory requirements for neighbor lists are shown for varying supports. It can be seen that, e.g., compressed neighbor lists for a support of $3.5h$ with 184 neighbors require less memory than uncompressed neighbor lists for a support of $2h$ with 30 neighbors as indicated in Fig. 8.

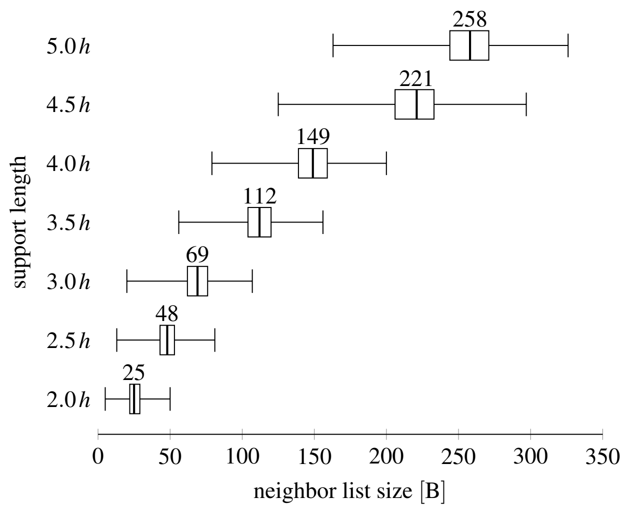


Figure 9: Memory requirements of the neighbor lists for the dam break scenario for various support lengths. The central mark and box margins are used in the same manner as in Fig. 8.

5.5. Scaling with the Number of Particles

Table 6 shows how our CLL approach scales with the number of particles. Therefore, we use the dam break scenario with varying resolutions resulting in 0.2, 11.6, 206.2 and 7440.9 million particles. We also use the wind tunnel scenario with 1302.1 million particles. Memory consumption and computation times for the neighbor lists and for entire simulation steps are given for three different hardware settings. We discuss three special settings.

Interactive SPH simulation on a 12*-core PC:* We are able to compute 25 simulation steps per real-time second for 0.2 million particles. DFSPH is used with two divergence-free iterations and two density-invariance iterations.

Large-scale SPH simulations on a single workstation: Our neighbor search and storage enables the simulation of 206.2 million SPH particles on a 12-core PC using only 28 GB of memory. Even more remarkable, 1302.1 million SPH particles can be processed on a 24-core PC using 172 GB of memory. Such particle number have previously been computed on, e.g., 64 GPUs [DCVB*13] or CPU clusters with 2560 cores [BKB16].

Large-scale on multiple workstations: By connecting multiple workstations, we are able to simulate the dam break scenario with a resolution that results in 7440.9 million particles. This setting requires 872 GB of memory without linear-exact kernel gradients. The performance is limited by the speed of our network and the unequally distributed amount of available memory per workstation. Nevertheless, we can simulate one step of the 200 million dam break scene in 15.254 s in this setting rather than 55.857 s using one 12-core PC.

Compressing the neighbor lists is particularly appropriate for scenes with a large particle numbers to capture small-scale details. This is illustrated in the wind tunnel scenario in Fig. 10 which is simulated on our 112-core setup. The total computation time per simulation step is 80.17 s on average whereof the computation time for searching and storing the neighbor lists is 8.95 s. A maximum of 210.8 GB of memory is required for the simulation.

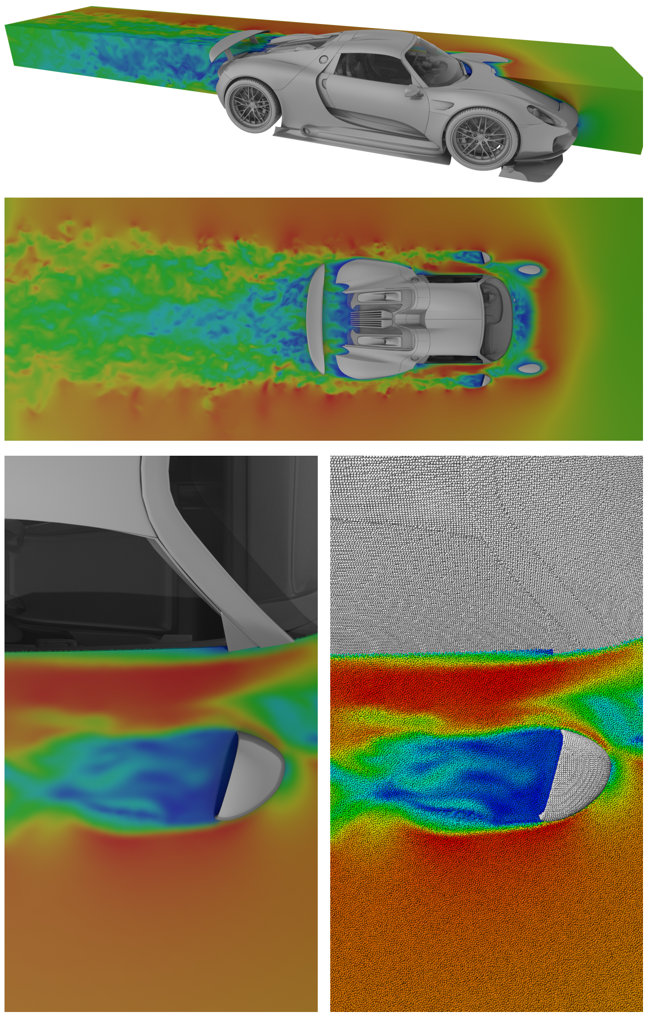


Figure 10: Large-scale wind tunnel scene with 1272 million fluid particles simulated on six workstations using the standard message passing interface (MPI). The bottom image shows a closeup, visualizing the particles with color-coded velocities.

6. Conclusion and Future Work

Neighbor computation and storage is often considered a critical aspect in SPH simulations. This paper contributes to improved data structures and algorithms in that topic with a specific focus on memory consumption. As neighbor lists are responsible for a large portion of the overall memory consumption, we propose to compress these lists with a novel scheme. Accompanied by a novel memory-efficient CLL variant, the overall memory consumption can be significantly reduced, e.g., by a factor of nine compared to the state-of-the-art as indicated in Table 4. The lean data structure of our CLL enables a straightforward implementation that is fully parallelized. While all scenarios generally require less memory with marginal computational overhead, it is particularly remarkable

			particles [million]		average computation time per time step [s]		memory [GB]
workstation	scene	particle size [mm]	fluid	boundary	neighbor list creation	total	maximum
12 core	dam break	15.50	0.15	0.05	0.005	0.039	0.12
12 core	dam break	4.00	10.5	0.8	0.287	2.747	1.87
12 core	dam break	1.50	200	6.2	5.378	55.857	28.13
112 core	dam break	1.50	200	6.2	2.022	15.254	32.52
112 core	dam break	0.45	7373	67.9	66.341	527.671	872.70
24 core	wind tunnel	4.00	1272	30.1	32.771	355.168	171.97
112 core	wind tunnel	4.00	1272	30.1	8.950	80.170	210.80

Table 6: Memory consumption and performance for two scenes with varying resolutions using three hardware settings. Our compression scheme and CLL scheme is used in all settings.

that SPH fluid simulations with about one billion particles can now be processed on a single 24-core PC using only 172 GB of memory. Such scenarios have been rarely computed previously, as they required larger GPU or CPU clusters.

GPU implementations are beyond the scope of the paper, but might be an interesting direction for future research. Since we are able to process 0.2 million particles with 25 simulation steps per real-time second on a 12-core CPU, it is certainly interesting to investigate the respective GPU capabilities. Another interesting research direction might be the re-consideration of Verlet lists. Such lists are updated only every few simulation steps, but are much larger than the lists of actual neighbors. As our compression scheme is particularly useful for larger lists, it can be interesting to investigate the potential of compressed Verlet lists.

Acknowledgments

We thank FIFTY2 Technology for providing an academic license of PreonLab [FIF19]. The car model is courtesy of primero-tag at https://www.cgtrader.com and is licensed under Royalty Free License.

References

[AAT13] Akinci N., Akinci G., Teschner M.: Versatile surface tension and adhesion for sph fluids. *ACM Transactions on Graphics 32*, 6 (2013), 182:1–182:8. 5

[AIA*12] Akinci N., Ihmsen M., Akinci G., Solenthaler B., Teschner M.: Versatile rigid-fluid coupling for incompressible sph. *ACM Transactions on Graphics 31*, 4 (2012), 62:1–62:8. 5

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. *ACM Transactions on Graphics 26*, 3 (2007). 3

[Bad12] Bader M.: *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer, 2012. 6

[BGI*18] Band S., Gissler C., Ihmsen M., Cornelis J., Peer A., Teschner M.: Pressure boundaries for implicit incompressible sph. *ACM Transactions on Graphics 37*, 2 (2018), 14:1–14:11. 1

[BGPT18] Band S., Gissler C., Peer A., Teschner M.: Mls pressure boundaries for divergence-free and viscous sph fluids. *Computers & Graphics 76* (2018), 37–46. 5

[BK17] Bender J., Koschier D.: Divergence-free sph for incompressible and viscous fluids. *IEEE Transactions on Visualization and Computer Graphics 23*, 3 (2017), 1193–1206. 1, 5

[BKB16] Braun S., Koch R., Bauer H.-J.: Smoothed particle hydrodynamics for numerical predictions of primary atomization. In *High Performance Computing in Science and Engineering '16* (2016), Springer, pp. 321–336. 8

[BL99] Bonet J., Lok T.-S. L.: Variational and momentum preservation aspects of smooth particle hydrodynamic formulations. *Computer Methods in Applied Mechanics and Engineering 180*, 1 (1999), 97–115. 5

[DCGG13] Domínguez J. M., Crespo A. J. C., Gómez-Gesteira M.: Optimization strategies for cpu and gpu implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications 184*, 3 (2013), 617–627. 2, 3, 4, 5, 6

[DCGGM11] Domínguez J. M., Crespo A. J. C., Gómez-Gesteira M., Marongiu J. C.: Neighbour lists in smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids 67*, 12 (2011), 2026–2042. 1, 3

[DCVB*13] Domínguez J. M., Crespo A. J. C., Valdez-Balderas D., Rogers B. D., Gómez-Gesteira M.: New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications 184*, 8 (2013), 1848–1860. 3, 8

[Dea09] Dean J.: Challenges in building large-scale information retrieval systems: Invited talk. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining* (2009), ACM, pp. 1–1. 2

[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*, 4 (2008), 531–553. 6

[FIF19] FIFTY2 Technology: PreonLab. `www.fifty2.eu`, 2019. 6, 9

[Fox88] Fox G. C.: A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *Numerical Algorithms for Modern Parallel Computer Architectures* (1988), Springer, pp. 37–61. 5

[Gan15] Ganzenmüller G. C.: An hourglass control algorithm for Lagrangian smooth particle hydrodynamics. *Computer Methods in Applied Mechanics and Engineering 286* (2015), 87–106. 5

[GBP*17] Gissler C., Band S., Peer A., Ihmsen M., Teschner M.: Generalized drag force for particle-based simulations. *Computers & Graphics 69* (2017), 1–11. 5

[GEF15] Goswami P., Eliasson A., Franzén P.: Implicit incompressible sph on the gpu. In *Virtual Reality Interactions and Physical Simulations* (2015), Eurographics Association. 3

[Gon07] Gonnet P.: A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry 28*, 2 (2007), 570–573. 3

[GPB*19] GISSLER C., PEER A., BAND S., BENDER J., TESCHNER M.: Interlinked sph pressure solvers for strong fluid-rigid coupling. *ACM Transactions on Graphics 38*, 1 (2019), 5:1–5:13. 1

[GR11] GAFTON E., ROSSWOG S.: A fast recursive coordinate bisection tree for neighbour search and gravity. *Monthly Notices of the Royal Astronomical Society 418*, 2 (2011), 770–781. 3

[Gre10] GREEN S.: Particle simulation using cuda. *NVIDIA whitepaper 6* (2010), 121–128. 3, 4

[GRLS18] GUO X., ROGERS B. D., LIND S., STANSBY P. K.: New massively parallel scheme for incompressible smoothed particle hydrodynamics (isph) for highly nonlinear and distorted flow. *Computer Physics Communications 233* (2018), 16–28. 6

[GRS98] GOLDSTEIN J., RAMAKRISHNAN R., SHAFT U.: Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering* (1998), IEEE Computer Society, pp. 370–379. 3

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive sph simulation and rendering on the gpu. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010), Eurographics Association, pp. 55–64. 3, 4

[HBMW11] HUBBER D. A., BATTY C. P., MCLEOD A., WHITWORTH A. P.: SEREN – a new sph code for star and planet formation simulations - algorithms and tests. *Astronomy & Astrophysics 529* (2011), A27. 3

[HCM06] HEGEMAN K., CARR N. A., MILLER G. S. P.: Particle-based fluid simulation on the gpu. In *Computational Science – ICCS 2006* (2006), Springer, pp. 228–235. 3

[HGE74] HOCKNEY R. W., GOEL S. P., EASTWOOD J. W.: Quiet high-resolution computer models of a plasma. *Journal of Computational Physics 14*, 2 (1974), 148–158. 1, 3

[HHM19] HUANG L., HÄDRICH T., MICHELS D. L.: On the accurate large-scale simulation of ferrofluids. *ACM Transactions on Graphics* (2019). 1

[Hil91] HILBERT D.: Über die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen 38*, 3 (1891), 459–460. 6

[HKK07] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Smoothed particle hydrodynamics on gpus. *Computer Graphics International 40* (2007). 3

[IABT11] IHMSEN M., AKINCI N., BECKER M., TESCHNER M.: A parallel sph implementation on multi-core cpus. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 99–112. 1, 2, 3, 4, 6

[ICS*14] IHMSEN M., CORNELIS J., SOLENTHALER B., HORVATH C., TESCHNER M.: Implicit incompressible sph. *IEEE Transactions on Visualization and Computer Graphics 20*, 3 (2014), 426–435. 1

[IOS*14] IHMSEN M., ORTHMANN J., SOLENTHALER B., KOLB A., TESCHNER M.: Sph fluids in computer graphics. In *Eurographics (State of the Art Reports)* (2014). 3

[KAG*05] KEISER R., ADAMS B., GASSER D., BAZZI P., DUTRE P., GROSS M.: A unified lagrangian approach to solid–fluid animation. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics* (2005), pp. 125–148. 3

[KFV*05] KLEEFSMAN K. M. T., FEKKEN G., VELDMAN A. E. P., IWANOWSKI B., BUCHNER B.: A volume-of-fluid based simulation method for wave impact problems. *Journal of Computational Physics 206*, 1 (2005), 363–393. 6

[KS09] KALOJANOV J., SLUSALLEK P.: A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 23–28. 4

[LB15] LEMIRE D., BOYTSOV L.: Decoding billions of integers per second through vectorization. *Software – Practice & Experience 45*, 1 (2015), 1–29. 2, 3

[LF80] LADNER R. E., FISCHER M. J.: Parallel prefix computation. *Journal of the ACM 27*, 4 (1980), 831–838. 3

[LKR17] LEMIRE D., KURZ N., RUPP C.: Stream vbyte: Faster byte-oriented integer compression. *Information Processing Letters 130* (2017), 1–6. 1, 2, 3, 4

[LSY01] LEE R., SHI Z., YANG X.: Efficient permutation instructions for fast software cryptography. *IEEE Micro 2* (2001). 4

[MFZ97] MORRIS J. P., FOX P. J., ZHU Y.: Modeling low reynolds number incompressible flows using sph. *Journal of Computational Physics 136*, 1 (1997), 214–226. 5

[MM13] MACKLIN M., MÜLLER M.: Position based fluids. *ACM Transactions on Graphics 32*, 4 (2013), 104:1–104:12. 1

[Mor66] MORTON G. M.: *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966. 4, 6

[MRR12] MCCOOL M., ROBISON A. D., REINDERS J.: Chapter 13 - merge sort. In *Structured Parallel Programming*. Morgan Kaufmann, 2012, pp. 299–305. 4

[MRSD15] MOKOS A., ROGERS B. D., STANSBY P. K., DOMÍNGUEZ J. M.: Multi-phase sph modelling of violent hydrodynamics on gpus. *Computer Physics Communications 196* (2015), 304–316. 3

[NAM10] NGOC ANH V., MOFFAT A.: Index compression using 64-bit words. *Software, Practice and Experience 40* (2010), 131–147. 2

[OD08] ONDERIK J., DURIKOVIC R.: Efficient neighbor search for particle-based fluids. *Journal of the Applied Mathematics, Statistics and Informatics 4*, 1 (2008), 29–43. 4

[OLTG*16] OGER G., LE TOUZÉ D., GUIBERT D., DE LEFFE M., BIDDISCOMBE J., SOUMAGNE J., PICCINALI J.-G.: On distributed memory mpi-based parallelization of sph codes in massive hpc context. *Computer Physics Communications 200* (2016), 1–14. 6

[OM84] ORENSTEIN J. A., MERRETT T. H.: A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (1984), ACM, pp. 181–190. 5

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2003), Eurographics Association, pp. 41–50. 4, 6

[PF01] PASCUCCI V., FRANK R. J.: Global static indexing for real-time exploration of very large regular grids. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (2001), ACM, pp. 2–2. 4

[PGBT18] PEER A., GISSLER C., BAND S., TESCHNER M.: An implicit sph formulation for incompressible linearly elastic solids. *Computer Graphics Forum 37*, 6 (2018), 135–148. 1

[PH10] PELFREY B., HOUSE D.: Adaptive neighbor pairing for smoothed particle hydrodynamics. In *Advances in Visual Computing* (2010), Springer, pp. 192–201. 3

[Phe08] PHEATT C.: Intel® threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298. 5

[PKL15] PLAISANCE J., KURZ N., LEMIRE D.: Vectorized vbyte decoding. *arXiv* (2015). 2, 3

[PT17] PEER A., TESCHNER M.: Prescribed velocity gradients for highly viscous sph fluids with vorticity diffusion. *IEEE Transactions on Visualization and Computer Graphics 23*, 12 (2017), 2656–2662. 1

[SGL09] SCHLEGEL B., GEMULLA R., LEHNER W.: K-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware* (2009), ACM, pp. 52–60. 5

[SGR*11] STEPANOV A. A., GANGOLLI A. R., ROSE D. E., ERNST R. J., OBEROI P. S.: Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (2011), ACM, pp. 317–326. 2

[SP09] SOLENTHALER B., PAJAROLA R.: Predictive-corrective incompressible sph. *ACM Transactions on Graphics 28*, 3 (2009), 40:1–40:6. 1

[Spr05] SPRINGEL V.: The cosmological simulation code gadget-2. *Monthly notices of the royal astronomical society 364*, 4 (2005), 1105–1134. 3

[TH72] THIEL L. H., HEAPS H. S.: Program design for retrospective searches on large data bases. *Information Storage and Retrieval 8*, 1 (1972), 1–20. 2

[TH81] TROPF H., HERZOG H.: Multimensional range search in dynamically balanced trees. *Angewandte Informatik 23* (1981), 71–77. 2, 4, 5

[TLTM18] TANG M., LIU Z., TONG R., MANOCHA D.: Pscc: Parallel self-collision culling with spatial hashing on gpus. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 1 (2018), 18:1–18:18. 3

[TSG14] THALER F., SOLENTHALER B., GROSS M.: A parallel architecture for iisph fluids. In *Virtual Reality Interactions and Physical Simulations* (2014), pp. 119–124. 5

[VBC08] VICCIONE G., BOVOLIN V., CARRATELLI E. P.: Defining and optimizing algorithms for neighbouring particle identification in sph fluid simulations. *International Journal for Numerical Methods in Fluids 58*, 6 (2008), 625–638. 3

[Ver67] VERLET L.: Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review 159* (1967), 98–103. 3

[Wen95] WENDLAND H.: Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics 4*, 1 (1995), 389–396. 5

[WKBB18] WEILER M., KOSCHIER D., BRAND M., BENDER J.: A physically consistent implicit viscosity solver for sph fluids. *Computer Graphics Forum 37*, 2 (2018). 1

[WMRR17] WINKLER D., MEISTER M., REZAVAND M., RAUCH W.: gpusphase – a shared memory caching implementation for 2d sph using cuda. *Computer Physics Communications 213* (2017), 165–180. 3

[WRR18] WINKLER D., REZAVAND M., RAUCH W.: Neighbour lists for smoothed particle hydrodynamics on gpus. *Computer Physics Communications 225* (2018), 140–148. 3

[WS95] WARREN M. S., SALMON J. K.: A portable parallel particle program. *Computer Physics Communications 87*, 1 (1995), 266–290. Particle Simulation Methods. 3

[WSG*18] WILLIS J. S., SCHALLER M., GONNET P., BOWER R. G., DRAPER P. W.: An efficient SIMD implementation of pseudo-verlet lists for neighbour interactions in particle-based codes. *arXiv* (2018). 3

[YDS09] YAN H., DING S., SUEL T.: Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web* (2009), ACM, pp. 401–410. 2

[ZHNB06] ZUKOWSKI M., HEMAN S., NES N., BONCZ P.: Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering* (2006), IEEE Computer Society, pp. 59–. 2

[ZZL*15] ZHAO W. X., ZHANG X., LEMIRE D., SHAN D., NIE J.-Y., YAN H., WEN J.-R.: A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems 33*, 3 (2015), 15:1–15:28. 2