EG UK Computer Graphics & Visual Computing (2019)
G. K. L. Tam and J. C. Roberts (Editors)

Fast and Efficient Nearest Neighbor Search for Particle Simulations

J. Groß[1], M. Köster[1] and A. Krüger[1]

[1]Saarland Informatics Campus, Germany

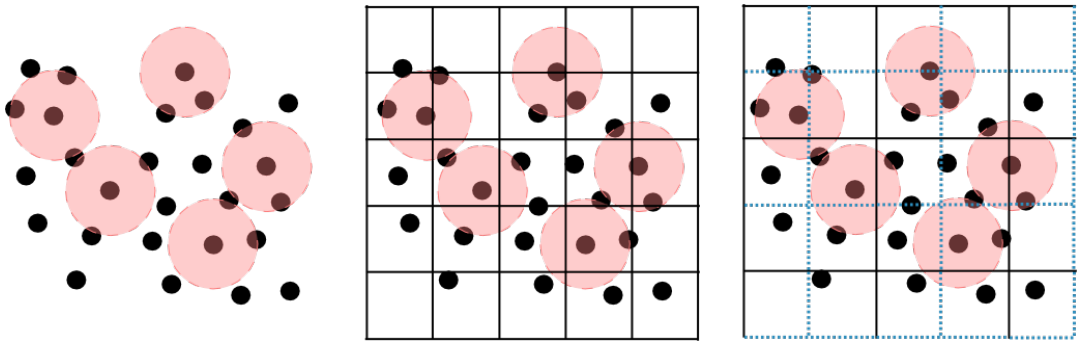


Figure 1: *A set of particles (left) and some highlighted neighbor-lookup radii. A commonly used data structure to speed up neighbor-lookup in general is a uniform grid that subdivides the space into several grid cells (middle). Our approach (right) uses two steps to subdivide the domain into uniform grid cells. The first step (black lines) forms a coarse-grained grid that is stored in memory. The fine-grained grid is created on-the-fly in the second step during neighbor lookup (blue lines) and does not need to be stored in GPU device memory. This dramatically reduces memory consumption and improves lookup times due to coherent memory accesses to global and shared memory.*

Abstract

One of the fundamental algorithms in particle simulations is the identification and iteration over nearest neighbors of every particle. Well-known examples are SPH or PBD simulations that compute forces and particle-position updates in every simulation step. In order to find nearest neighbors for all particles, hash-based, grid-based or tree-based approaches have been developed in the past. The two most prominent and fastest algorithms use virtual and explicitly allocated uniform grids to achieve high performance on Graphics Processing Units (GPUs). However, they have disadvantages with numerous particle simulation domains, either in terms of run time or memory consumption. We present a novel algorithm that can be applied to large simulation domains that significantly reduces memory consumption using a shared-memory based neighbor search. Furthermore, we achieve high-performance on our evaluation scenarios that often outperforms existing state-of-the-art methods.

CCS Concepts

• ***Computing methodologies*** → ***Shared memory algorithms; Massively parallel algorithms;*** *Graphics processors;*

1. Introduction

Particle simulations are very useful and often an essential utility to invent or evaluate mathematical approaches in simulated environments. Prominent examples are *Smoothed Particle Hydrodynamics* (SPH) [GSSP10], *Position Based Dynamics* (PBD) [MHHR07] or *Position Based Fluids* (PBF) [MM13, KK16]. SPH uses different kinds of so called smoothing kernels that calculate various forces between particles. Similar to SPH, PBF also relies on the evaluation of smoothing kernels to compute local fluid densities. PBD leverages constraints to model the interaction behavior between particles using their position, velocity and mass information. All these approaches use an iterative solver to approximate the final solution. The iteration over all neighboring particles is required in every solver step, and thus often dominates the overall run-time performance.

A similar problem also appears in the field of computer graphics. *Photon mapping* shoots many photon particles from all light sources into the scene being rendered. During the actual rendering

DOI: 10.2312/cgvc.20191258

process, nearest neighbor search is required to gather the k-nearest photons in order to approximate the illumination of a particular position in the scene. Again, a large portion of the run-time performance depends on the gathering step that has to be performed several times for every pixel in the rendered image. In contrast to the previously presented domain, an approximation of the k-nearest particles is often sufficient, since there is typically only a little loss in terms of image quality if we skip some neighbors. This domain knowledge can be exploited to create fast and efficient search algorithms. Gupte [Gup12] , Pedersen [Ped13] and Fleisz [Fle09] show interesting and promising approaches for hashing these photons in a scene to improve run-time performance.

We are focusing on a non-approximative neighborhood search that considers all neighboring particles. Skipping some particles for performance reasons can lead to non-deterministic behavior in terms of simulation correctness. For practical considerations, all particles in a certain fixed-size radius around every particle are considered to be neighboring particles in order to avoid useless calculation overhead; like for example in [KE10], [BVAT17] and [KSG15, KK18]. For this reason, several approaches have been developed using either grid-based or tree-based algorithms. In the past, tree-based approaches were widely spread until the significantly faster state-of-the-art grid-based methods have been introduced on GPUs [Gre08, Hoe13]. The grid-based methods are more suitable for these problems and much faster and easier to implement on GPUs, since they ensure coherent memory accesses and use static memory allocations. However, major disadvantages of these methods are the large memory consumption of $O(m^3)$, where m is the grid size in one dimension.

In this paper, we propose a new algorithm that has significantly reduced memory consumption using fast on-chip *shared memory*. We leverage recent improvements of GPU-hardware in terms of fast intrinsic functionality to improve performance in comparison to the current state-of-the-art methods. Furthermore, our approach is designed to benefit from coherent memory accesses to large portions of the underlying particle data. We demonstrate that our approach needs almost no additional memory but still has a high performance on our evaluation scenarios.

2. Related Work

There are mainly three concepts to determine nearest neighbor particles: straight-forward, grid-based and tree-based approaches. The simplest approach is a brute-force method, where all particles are checked, whether they are in range of a certain particle or not. This is very inefficient especially for domains with lots of particles, since the lookup has a run-time complexity of $O(n^2)$. A common use case of this method is the N-body problem which computes gravitational forces between all objects [KSG15]. However, more advanced algorithms like the Barnes-Hut algorithm [BP11, KSC*14] use trees to reduce processing complexity.

Generic tree-based approaches use a construction phase to create a data structure with the overall purpose of decreasing lookup time. The disadvantage of these trees on a GPU is their dynamic manner since tree construction needs dynamic memory allocations to insert all particles. These memory allocations during run time needs long time and therefore should be avoided. A static memory allocation before running the algorithm is much more efficient, but makes the usage of trees more sophisticated. However, tree-based approaches are commonly used to improve run time and memory consumption to build the neighborhood relation between particles. Otair [Ota13], Gieseke [GHOI14], Zhou [ZHWG08], Goradia [Gor08] and Qui [QMN09] show different approaches to handle efficient nearest neighbor queries with kd-trees. The original idea of a kd-tree construction can be followed back in the 70's where Bentley introduced this method [Ben75]. An additional downside of trees on GPUs are their (often) not optimal memory access patterns, in general. Maintaining the data structure over time requires either sophisticated algorithms or a complete reconstruction in every simulation step.

The two currently fastest approaches use grid structures to improve nearest neighbor search. The first approach from Green [Gre12] uses a virtual grid, which hashes particles to certain grid cells. These cells represent a certain part of the simulation space. After the initial insertion phase, all particles are reordered using *radix sort* [SHG09] to improve coherent memory accesses. Later on, these grid cells (as well as their neighboring cells) are looked up for potentially neighboring particles. This concept mainly suffers from run-time limitations imposed by the *radix sort* algorithm, which often performs slowly if we have to consider lots of particles. The second state-of-the-art approach is the one from Hoetzlein [Hoe13]. It was developed later on and improves Green's approach by using *atomic functions*. These functions, which represent hardware-based synchronized memory accesses, can be used to increment counters in global memory. Such a function returns the old value of the memory cell that (in this case) corresponds to the particle offset within a certain cell. This value is used as a cell offset in the reordering step that is added to the start index of a grid cell. Hoetzlein uses a large grid, where each memory cell represents a grid cell containing its actual number of particles. A *prefix sum* [MG16] algorithm calculates the offsets for each grid cell. Afterwards, all particles are reordered in a separate step to move them to their final position in memory. This significantly improves run-time performance at the expense of a high memory consumption.

Kawada [KGIN11] invents an idea in which they add some "bounding-space" around each grid cell to eliminate an additional lookup table. However, this leads to duplicate particles as a particle that falls into several boundary-regions needs to be present in all of them. The result is a higher memory consumption due to storing of additional interim results and involves an additional duplicate-elimination step.

3. Algorithm

Hoetzlein and Green provide simple approaches to sort particles, that do not use much hardware-related features which can improve run time and memory consumption. The only hardware-related feature is the use of *atomic functions* in Hoetzlein's version to increment cell counters in global device memory. However, increasing the number of particles or the cell sizes leads to larger grids and many atomic-function accesses to the same memory addresses. Larger grids imply an increase of scattered memory accesses to arbitrary locations. Furthermore, many atomic-function operations

Figure 2: *A grid cell that is divided in its inner region (red) and the border region (green). The width of the border region is equal to the search radius.*

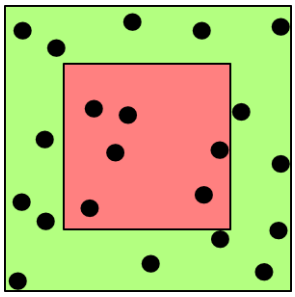
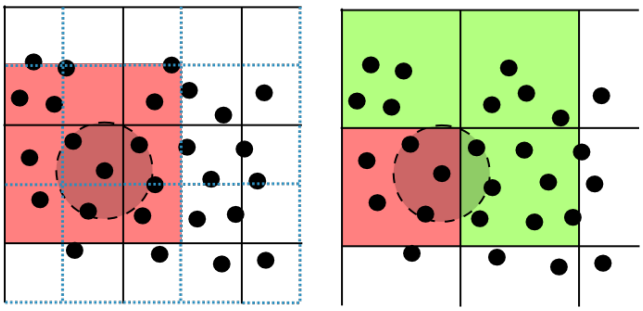


Figure 3: *A particle and its associated search radius falls into a grid cell of the second step (left). From a theoretical point of view, we have to check all marked grid cells for neighbors (red). However, since we do not explicitly store the grid of the second step in memory, we have to check all potential neighbors from the border part of the other cells resulting from the first step (right, green).*



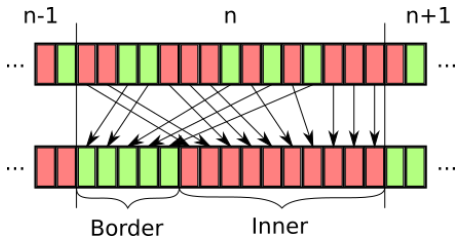
on the same memory locations often result in severe slowdowns (due to serialization effects). Using fast on-chip *shared memory* in a smart way can speed up the overall neighbor search. The general idea of our method is the usage of small grids that still fit into shared memory. In this case, we can use fast atomic functions to update memory cells in shared memory in the scope of a single thread group [NVI19].

Unfortunately, shared memory is still quite limited in terms of size [NVI19]. Storing a single grid in shared memory, therefore, leads to a small grid resolution and large grid cells containing hundreds of particles. This does not improve performance, since we have to iterate over all particles in the same cell (and all neighboring cells). Our idea is a general two-step approach. In the first step, we subdivide the domain into a low-resolution grid that still fits into shared memory and leverage Hoetzlein's cell-counter principle (see subsection 3.1). Since we want to process all particles in parallel, we have to launch lots of thread groups. However, these groups cannot communicate via shared memory directly, we still need some global synchronization step afterwards. Following Hoetzlein's idea, we also perform a global prefix sum (like in [MG16]), such that we can obtain the global start offset for each grid cell from the first step. This means that we still need the small grid in global memory to calculate the offsets and synchronize the results.

In the second step, we subdivide every cell from the first step again into a low-resolution grid that still fits into shared memory of every thread group. Note that the grid resolution in the second step is typically lower than in the first step, since we have to store additional information in shared memory (see subsection 3.2). In contrast to the first step, each thread group processes a single grid cell and its containing particles. During the actual neighbor search, the conceptionally fine-grained grid is locally constructed on-the-fly in shared memory. This allows us to efficiently iterate over neighboring particles within the same grid cell. However, this does not help when iterating over potential neighbors from other grid cells of the first step.

To solve this issue we introduce the concept of a *border region* of every grid cell. Consider the particles in Figure 2. We split up the grid cell in two sub regions. The green area represents the border region that contains all particles that may be looked up by neighboring grid cells for potential neighbors. The size of this region depends on the chosen search radius. For practical considerations, the search radius should chosen in a such a way that we only need to check the 27 neighboring grid cells in 3D. The red area contains particles that cannot be looked up by particles in other grid cells. While Hoetzlein and Green check all neighbor cells with all particles, we only check the potentially relevant particles in the border region of other grid cells from the first step. Hoetzlein uses huge grids to minimize look-up overhead, because he has only some particles in this grid region to look up. We leverage border regions since we have only low-resolution grids where each grid cell can contain lots of particles (see Figure 3).

Figure 4: *Memory layout of particles in the scope of an inner region (red) and a border region (green). We move particles that fall into the border region to the start address. This is not a strict requirement, as it is only important that one section contains the border particles and the other one the remaining particles. In our implementation the border region is filled from the left, whereas the inner region is filled from the right-hand side.*



3.1. First Step: Sorting

For sorting, we can modify Hoetzlein's approach to adapt the method to our needs. In contrast to his method, we have to sort the particles within a grid cell as outlined in Figure 4. Fortunately, we can do that in a fast and efficient way using an additional step during particle reordering. As a result, we get two sections per grid cell, where one section contains the border elements and the other section the inner ones. It is not important which section comes first, but we have to store an additional index pointing to the start location of the second section.

The actual sorting algorithm consists of two parts. The first part is shown in algorithm 1 and works as follows: In a first step, we calculate the grid position for each particle and increment a counter in shared memory with an *atomic addition*. After that, we store the local counter result in the global memory, also using an atomic operation. If all particles are processed and the results are written, we perform a prefix sum over the global-memory buffer, analogous to Hoetzlein's approach. We have to compute inclusive and an exclusive prefix-sum results since these offsets are required in the next part of the algorithm. Hence, the counter array needs to be twice as large, because we have to store both results in the same buffer. Hoetzlein uses two arrays to store the grid-cell index and the grid-cell offset of a particle, which increases memory consumption. These temporary memory arrays are not required in our case, because we can calculate the grid-cell position and use the results from the prefix sum to infer the offset.

The second part is shown in algorithm 2. Initially, we calculate the grid-cell position again. We compute whether the particle lies in the border region or not, where the result of the check is 0 or 1. This value is now added to a local counter in shared memory to track how many border particles exist in the current cell. Then, the offset is obtained from the global counter by an atomic add. In this case, the global memory access is needed, because we require the globally unique offset value of the current grid cell. The final position within a grid cell is calculated in the following way: If it is a border element we add the particle to the beginning of the reserved memory space (filled from left to right). The other particles are added to the end of the memory space, that means we fill from right to left. We need to add the local offset to the exclusive prefix sum result, if it is a border particle. Alternatively, we need to subtract it from the inclusive prefix sum result otherwise. After reordering, both prefix-sum parts contain the same values and we have to subtract the local border-particle counter again from one part to resolve the start indices of every grid cell.

Algorithm 1: The Sort Algorithm (Part 1)

```
   Input: index idx, input buffer input, array count, grid grid
1  sharedGrid := shared memory [maximum grid size];
2  foreach particle ∈ input do
   |  /* Calculate gridcell index                    */
3  |  pos := CalculateGridPosition (particle, grid);
4  |  atomicAdd (sharedGrid[pos], 1);
5  end
6  group barrier;
7  foreach (value, cellIdx) ∈ sharedGrid do
   |  /* Store accumalated value in global
   |     memory                                       */
8  |  atomicAdd (count[cellIdx], sharedGrid[cellIdx]);
9  end
```

Algorithm 2: The Sort Algorithm (Part 2)

```
   Input: index idx, input buffer input, output buffer output,
          array count, grid grid
1  sharedGrid := shared memory [maximum grid size];
   /* Divide the count-buffer into two
      parts: grid-cell start indices and
      inner-region start indices                    */
2  partSize := length(count) » 1;
3  foreach particle ∈ input do
   |  /* Calculate gridcell index                    */
4  |  pos := CalculateGridPosition (particle, grid);
   |  /* Check if the particle is in border
   |     region                                       */
5  |  isBorder := CheckBorderRegion (particle, grid);
6  |  atomicAdd (sharedGrid[pos], isBorder);
   |  /* Update corresponding index in
   |     count                                        */
7  |  reorderedPos := atomicAdd (count[pos + isBorder *
   |   partSize], isBorder * 2 - 1);
   |  /* Adjust target position in case of
   |     an inner-region particle since we
   |     fill the inner region from the
   |     right-hand side                              */
8  |  reorderedPos := reorderedPos + (isBorder - 1);
9  |  output[reorderedPos] := particle;
10 end
11 foreach (value, cellIdx) ∈ sharedGrid do
12 |  atomicAdd (count[cellIdx], -value);
13 end
```

3.2. Second Step: Neighbor Search

For neighbor search (algorithm 3) we need the reordered particles from the previous step and the buffer that contains the grid-cell start indices and the border-section start indices. The search is mainly done in shared memory, if possible, but some global memory accesses are needed due to space limitations (see Figure 5). We consider a neighbor iterator that consists of two functions: one is invoked for every neighboring particle and the other is invoked when all neighbors have been finished. This enables modeling of arbitrary neighbor iterators.

The algorithm is started for every global grid cell from the first step, which is represented by one GPU group. At first, we have to calculate a smaller local grid to form the acceleration structure for a particular grid cell. The 3D position of the minimum corner is obtained from the global grid cell and the neighbor-search radius. In the end, we have a grid which overlaps a global grid cell by the search radius in every direction. As grid dimension we choose a smaller dimension than the global grid, because we need to store additional data in shared memory. We reserve shared memory for managing the grid-cell offset counters of the local grid. Then, we have to resolve the start- and end index of the current global grid cell particles. Because we need group-wide barriers and, therefore, have to avoid thread divergences, we pad the number of particles to the next multiple of the group size.

The main loop processes all particles and fills up the padding area with some dummy particles that do not have any effect on the result. In this way we always process a subset of all particles in the cell in parallel. Afterwards, the grid offsets are reset to prepare the next iteration step. Similar to the first algorithm step from sub-

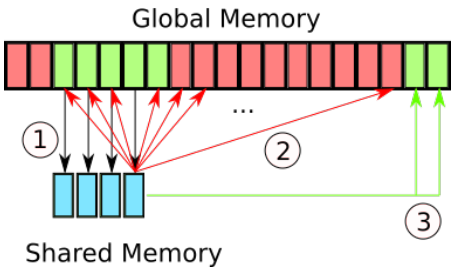
Algorithm 3: The Search Algorithm

```
   Input: input buffer input, array count, array innerLookup,
          array outerLookup, grid localGrid, grid globalGrid
1  particles := shared memory [group size];
2  gridOffsets := shared memory [size(localGrid)];
3  localGrid.Position := Calculate3DPositionOfLocalGridCell;
4  start, end := GetStartAndEndFromCount;
5  paddedEnd := ⌈(end - start) / group size⌉ * group size + start;
6  for i := start + group index; i < paddedEnd; i += group
     size do
7      particle := i < end ? input[i] : Vector3(float.MinValue);
8      pos := CalculateGridPosition (particle, localGrid);
9      reset gridOffsets;
10     group barrier;
11     if pos >= 0 then
12         offset := atomicAdd (gridOffsets[pos], 1);
13     end
14     group barrier;
15     prefix sum (gridOffsets);
16     if pos >= 0 then
17         targetPos := Exclusive(pos, gridOffsets) + offset;
18         particles[targetPos] := particle;
19     end
20     group barrier;
21     for j := start + group index; j < end; j += group size
         do
22         neighborParticle := input[j];
23         LookupNeighborCells(...);
24     end
25     group barrier;
26     if warp index < #neighboring cells then
27         outerPos := grid index + outerLookup[warp
             index];
28         if outerPos >= 0 & outerPos < size(globalGrid)
             then
29             startN, endN := GetStartAndEndFromCount;
30             startN := startN + lane index;
31             for k := startN; k < endN; k += warp size do
32                 LookupNeighborCells(...);
33             end
34         end
35     end
36     group barrier;
37     if i < end then
           /* Invoke the user-defined
              finish-iteration function     */
38         InvokeFinishNeighborIterationFunction;
39     end
40     group barrier;
41 end
```

Algorithm 4: Lookup Neighbor Cells Method

```
   Input: input buffer input, array gridOffsets, array
          innerLookup, particle neighborParticle, grid grid
1  pos := CalculateGridPosition (neighborParticle, grid);
2  for i := 0; i < #neighboring cells; i := i + 1 do
       /* Resolve the actual lookup position
          */
3      lookupPos := innerLookup[i] + pos;
4      if lookupPos >= 0 & lookupPos < size(grid) then
5          start := Exclusive(gridOffsets, lookupPos);
6          end := Inclusive(gridOffsets, lookupPos);
7          for k = start; k < end; k := k + 1 do
8              o := input[k];
9              oPos := CalculateGridPosition(o, grid);
               /* Invoke the user-defined
                  iteration function          */
10             InvokeNeighborIterationFunction;
11         end
12     end
13 end
```

Figure 5: *The basic overview of the search algorithm. First, some elements are copied from global to shared memory (black arrows). Then (step 2), each element in shared memory looks up the elements from its own grid cell and checks the neighborhood relation (red arrows). In step 3, the elements in shared memory check the border particles from the grid cells around their current cell (green arrows). After step 3, the next elements from the current grid cell are copied to shared memory.*



section 3.1, we locally sort all particles in the scope of the local grid. Once the particles are sorted, we benefit from the acceleration structure, which improves performance. The particle position in the local grid is calculated and we increment the local counter with an atomic addition and store the offset of a particle in the local grid cell. Once a subset of all particles has been processed, we perform a prefix sum over the counter array. Using the prefix-sum information, we can calculate the locally reordered position of a particle and store the particle information in our reserved shared-memory array.

Now, the actual search process starts and is divided in two stages. In the first stage, we look for neighbor particles within our grid cell: We check all neighboring local grid cells for potential neighbors of every loaded particle. For this purpose, we use an additional algorithm *LookupNeighborCells* in algorithm 4. However, the bor-

der particles in the neighboring global grid cells are still missing and have to be checked in the second stage. To provide more parallelism, we assign each neighbor cell to a warp and process the neighbors in parallel. Then, every warp checks, whether there are particles in the border section or not. In the case of border particles, we iterate over all of them and invoke the user-defined neighbor-particle iteration function.

Once all neighboring cells have been processed and our thread has a valid particle (no dummy element), we invoke the user-defined finish-iteration function.

3.3. Implementation Details

We have implemented our algorithm and all related methods in C# using the ILGPU-compiler [†] for all GPU programs (referred to as kernels). The usage of grid-stride loops in algorithms 1 and 2 ensures a better occupancy of the GPU, and therefore a faster run time during the sorting process. All presented algorithms are directly modeled via GPU kernels.

Because the actual memory consumption is known before running the algorithm, all memory buffers can be allocated before execution. In this way, we can avoid unnecessary dynamic memory allocations during run time. Our implementation of the prefix sum uses warp shuffles to improve performance [NVI14]. Note that we have to pad the number of elements to process in the scope of the loop in algorithm 3 in line 6 to the next multiple of the group size. These dummy particles are used for performance reasons only and are omitted after the search step.

4. Evaluation

4.1. Scenarios

For evaluation, we compare our approach with Hoetzlein's and Green's method on different scenarios in terms of run-time speed (in milliseconds) which can be found in tables 1, 2, 3. We choose the number of particles and the search radii to point out the difference between all methods. The search radii vary from 0.2 to 1.0 in scene dimensions, to reflect common use cases in which 27 neighboring grid cells have to be tested for potential neighbors. In all cases, we uniformly distribute all particles in the simulation domain. We have chosen this strategy to emulate a subset of a dense-particle simulation, in which particles are (more or less) evenly distributed across the simulation domain. For instance, a search radius of 1.0 in scenario 1 with 4,194,304 particles corresponds to a average number of 148 neighbors per particle.

We follow Hoetzlein's computation of the cell size from his implementation, which results in $\text{cellSize} = \frac{2 \cdot \text{neighbor radius}}{\text{grid density}}$, where grid density is set to be 1.0 in our simulations. Therefore, the actual grid dimension m is given by $m = \frac{\text{scene size}}{\text{cellSize}}$. We apply this formula to Hoetzlein's and Green's approach in tables 1 and 3. Our method chooses the largest possible grid size that still fits into shared memory: an $18 \times 18 \times 18$ grid in the first step. In terms of scene size, we set the scene-cell size to 10 (in scenarios 1, 2) in world coordinates. Scenario 3 uses a doubled scene-cell size of 20. In this case, we can start measurement only with a search radius of 0.3 (instead of 0.2 like in the two other scenarios); we exceed global-memory limitations otherwise. Detailed information about the used grid sizes of all methods can be found in Table 4.

We consider all particles to be unsorted in memory, so we simulate the case in which we are in the first iteration of a simulation (with randomized placed particles, for example). This represents the worst-case for all methods, since all approaches cannot benefit from a previously constructed particle ordering in memory. The run time is worse because we need a larger number of rearrangements of particle data since lots of particles are not in the right memory region. Increasing the number of particles increases this effect.

4.2. Run Time

We used two GPUs from NVIDIA: a GeForce GTX Titan X and a GeForce GTX 1080 Ti. Every performance measurement is the median execution time of 100 algorithm executions. Consider scenario 1, in which we use different grid dimensions depending on the search radii for both other approaches. In their cases, larger search radii perform much faster since the number of scattered reads can be considerably decreased. In our approach, the run time increases with larger search radii, which is caused by more look-ups in the border region (it contains more particles that have to be checked). In most cases, our approach outperforms the other methods (max. speedup of factor 374 in edge cases of search radius 0.2). However, in some cases we only reach a competitive performance resulting in a worst slowdown of factor 3.6. Scenarios, in which particles are non-uniformly distributed, often result in a comparable performance, but do not perform worse to our experience.

If we use the same grid size for all three approaches in scenario 2, we can see that the run time is independent from the search radius in Hoetzlein's and Green's approach, because particle look-ups need the same amount of time. If we increase the amount of particles, the run time increases dramatically in both approaches. The underlying grid is not optimized for this amount of particles, since the grid size should be significantly larger to achieve optimal performance. Our approach is more suitable for the underlying grid in this case. The neighbor-checking overhead per grid cell is hidden by the usage of our shared-memory grid structure.

A run-time comparison of scenario 1 and scenario 3 results in a similar behavior. Hoetzlein and Green have much overhead if they have to use large grids in the last scenario. Increasing the number of particles makes the problem more suitable for both and the run time improves. Our approach works better in the greater scene than in the smaller scene. This results from smaller border regions, that have to be checked in the larger scene.

4.3. Memory Consumption

If we compare the memory consumption of the three provided algorithms, we can see some differences. Green uses for sorting only a virtual grid that is not allocated in memory, but we need additional memory for sorting pairs (containing grid-cell and particle indices),

† www.ilgpu.net

Parameter		Hoetzlein				Green				FENNS			
#Elem	r	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ
1024	0.2	62.84	27.03	97.71	15.71	3.64	1.26	4.15	2.9	0.41	9.37	0.63	1.77
*	0.5	4.20	0.00	6.44	0.00	1.88	0.02	0.96	0.06	0.48	0.00	0.64	0.00
*	0.8	1.15	0.00	1.72	0.00	1.89	0.03	0.95	0.05	0.51	0.00	0.64	0.00
*	1.0	0.73	0.00	0.89	0.00	1.48	9.21	0.73	0.02	0.51	5.08	0.65	0.00
8192	0.2	65.28	5.26	98.45	0.52	3.88	6.75	4.20	0.03	1.18	5.53	1.72	1.08
*	0.5	4.40	4.20	6.51	0.76	1.92	7.34	1.06	0.00	1.18	0.00	1.78	0.00
*	0.8	1.31	6.82	1.71	1.06	1.94	8.42	0.86	1.67	1.28	7.34	1.87	1.33
*	1.0	0.77	4.74	0.96	0.06	1.63	10.44	0.74	0.01	1.29	5.32	1.91	0.00
65536	0.2	64.77	4.51	98.80	0.74	5.20	1.52	5.84	10.88	1.78	8.43	2.76	1.37
*	0.5	4.70	6.27	6.88	1.05	2.49	9.27	1.73	1.61	1.92	7.73	3.00	1.43
*	0.8	1.38	6.98	1.90	1.04	1.81	9.44	1.18	1.64	1.97	6.76	3.08	1.30
*	1.0	0.76	3.34	1.12	1.08	1.99	9.72	0.97	1.73	1.99	8.86	3.12	1.32
524288	0.2	71.02	19.62	105.05	4.80	13.82	4.65	16.62	0.65	3.16	1.10	4.68	1.43
*	0.5	7.61	6.06	9.74	1.92	4.52	8.12	6.04	1.38	3.52	8.33	5.22	1.40
*	0.8	4.00	5.03	4.65	1.05	3.29	6.20	5.02	1.22	4.16	3.05	6.32	1.77
*	1.0	3.47	5.38	3.79	1.03	3.25	8.28	4.73	1.27	4.39	5.03	6.66	1.34
4194304	0.2	90.21	17.80	131.92	3.19	41.12	2.38	53.62	2.64	17.16	1.96	29.54	1.31
*	0.5	29.13	5.67	36.72	1.06	26.27	1.15	38.34	5.70	24.47	7.98	41.06	1.24
*	0.8	27.97	4.30	34.37	1.43	29.55	1.77	43.73	2.16	31.22	7.25	52.17	1.25
*	1.0	27.13	7.77	33.95	1.95	29.25	1.46	43.29	0.18	35.29	6.94	59.71	1.22

Table 1: *Scenario 1. Scene size: 180 x 180 x 180. Grid size of FENNS: 18 x 18 x 18, other: see Table 4. Times in ms.*

Parameter		Hoetzlein				Green				FENNS			
#Elem	r	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ
1024	0.2	0.18	5.48	0.12	2.07	1.14	11.91	0.56	2.76	0.52	8.74	0.69	1.80
*	0.5	0.25	0.00	0.12	0.00	0.96	0.06	0.56	0.00	0.57	0.00	0.63	0.00
*	0.8	0.24	0.00	0.12	0.00	1.69	0.02	0.59	0.00	0.62	0.00	0.65	0.00
*	1.0	0.26	0.00	0.12	0.00	1.09	0.13	0.57	0.00	0.60	0.00	0.65	0.00
8192	0.2	0.23	0.00	0.15	0.00	1.62	0.06	0.59	0.00	1.10	0.00	1.70	0.00
*	0.5	0.23	0.00	0.17	0.00	1.00	0.02	0.59	0.01	1.12	0.00	1.79	0.00
*	0.8	0.24	5.82	0.15	0.00	1.42	9.89	0.63	0.01	1.50	4.41	1.86	0.00
*	1.0	0.23	0.00	0.15	0.00	1.05	0.09	0.58	0.00	1.25	0.00	1.90	0.00
65536	0.2	0.51	0.00	0.60	0.00	1.41	2.66	1.10	0.01	1.76	5.45	2.75	0.00
*	0.5	0.48	0.00	0.60	0.78	1.55	0.07	1.02	0.00	1.89	4.53	2.99	0.00
*	0.8	0.52	0.00	0.61	0.84	1.63	0.04	1.02	1.72	1.94	0.00	3.07	1.00
*	1.0	0.52	3.22	0.59	0.00	1.42	8.29	1.01	0.01	1.96	0.00	3.10	0.00
524288	0.2	7.45	5.00	12.52	0.74	8.33	2.30	14.36	0.01	3.14	5.69	4.60	1.00
*	0.5	7.47	0.00	12.56	0.76	8.35	0.00	14.41	1.40	3.52	0.00	5.16	1.04
*	0.8	7.48	4.48	12.64	0.76	8.40	9.63	14.50	1.61	4.18	6.86	6.31	1.05
*	1.0	7.50	0.00	12.66	0.72	8.40	11.88	14.54	1.33	4.39	6.46	6.66	1.09
4194304	0.2	419.90	5.73	770.08	80.75	426.09	8.36	779.50	1.12	17.28	4.11	30.31	2.73
*	0.5	418.28	6.04	770.05	164.4	426.14	6.71	779.66	0.08	24.86	7.68	41.57	3.11
*	0.8	418.28	3.74	770.14	198.4	426.26	2.20	779.72	0.07	31.53	1.74	52.83	3.62
*	1.0	418.79	19.17	769.74	206.1	426.23	2.49	779.29	0.09	35.67	2.73	59.95	3.9

Table 2: *Scenario 2. Scene size: 180 x 180 x 180. Grid size of all methods: 18 x 18 x 18. Times in ms.*

which needs one 64bit integer per particle. To get the start indices of a grid cell, we additionally need an int memory array with length of the grid size. This leads to $8 \cdot n + 4 \cdot m^3$ bytes of additional memory for this approach, where n is the number of particles and m the grid-size in one dimension. Hoetzlein uses the same amount of memory for additional particle information, because he stores the grid-cell index and the particle offset within its grid cell resulting in an 64bit integer per particle. For the start offsets per grid cell, he needs m^3 times more space. So Hoetzlein also uses $8 \cdot n + 4 \cdot m^3$ bytes of additional space, if the prefix-sum computation can be performed in-place. If the GPU architecture does not allow such an implementation, we need additional $4 \cdot m^3$ memory which results in a total memory consumption of $8 \cdot n + 4 \cdot 2 \cdot m^3$. Especially in the case of large grids, the two approaches can easily exceed the memory limit. In the first scenario, a search radius of 0.1 is not possible since memory allocation on the GPU fails. The same holds for a search radius of 0.2, if we double the scene dimensions.

Our approach needs significantly less memory: We need $8 * l^3$ bytes in global memory, where $l << m$ and to our experience

Parameter		Hoetzlein				Green				FENNS			
#Elem	r	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ	1080 Ti	σ	Titan X	σ
1024	0.3	148.72	36.09	231.73	6.15	6.24	2.28	8.53	15.66	0.47	8.37	0.61	1.99
*	0.5	32.19	4.13	50.29	0.00	2.59	0.03	2.48	0.06	0.52	0.00	0.62	0.00
*	0.8	7.92	0.18	12.26	0.00	1.29	0.03	1.20	0.00	0.46	0.00	0.64	0.00
*	1.0	4.13	0.00	6.37	0.07	1.66	0.01	0.97	0.05	0.49	0.00	0.64	0.00
8192	0.3	152.71	0.28	232.60	0.00	6.37	0.02	8.94	0.04	1.10	0.00	1.70	0.00
*	0.5	33.43	3.92	50.47	2.54	0.09	0.00	2.49	0.02	1.77	0.00	1.76	0.00
*	0.8	8.25	0.00	12.34	0.00	1.74	0.02	1.33	0.08	1.22	0.00	1.82	0.02
*	1.0	4.27	0.00	6.41	0.00	1.70	0.10	1.06	0.01	1.23	0.00	1.83	0.00
65536	0.3	154.82	0.00	235.57	0.00	8.07	0.02	10.18	0.12	1.75	0.00	2.71	0.00
*	0.5	33.98	5.62	51.70	0.00	4.28	0.02	4.08	0.03	1.83	0.00	2.86	0.00
*	0.8	8.89	1.25	12.92	0.00	2.98	3.92	2.35	0.05	1.90	6.45	3.00	0.00
*	1.0	4.72	0.00	6.83	0.86	2.48	0.01	1.83	0.01	1.91	0.00	3.04	0.00
524288	0.3	160.10	4.13	242.20	0.26	18.52	7.53	22.71	2.53	3.12	0.66	4.59	1.01
*	0.5	38.11	3.36	56.05	2.49	10.47	1.29	12.76	3.86	3.20	0.73	4.71	0.99
*	0.8	11.27	0.00	15.65	0.65	5.51	6.06	7.30	1.61	3.31	6.14	4.89	1.06
*	1.0	7.07	3.72	9.57	0.67	4.49	9.36	6.12	1.20	3.49	6.64	5.14	0.97
4194304	0.3	186.09	2.35	272.97	0.89	55.21	0.70	70.78	0.15	15.57	1.55	27.00	0.96
*	0.5	58.60	0.00	82.72	1.71	30.20	1.09	42.80	2.25	18.03	0.00	30.61	0.94
*	0.8	32.70	0.82	43.26	0.13	26.22	1.20	38.04	0.14	21.62	2.10	36.50	0.97
*	1.0	29.00	2.56	36.77	0.16	26.43	1.47	38.47	0.17	23.87	1.18	40.12	1.44

Table 3: *Scenario 3. Scene size: 360 x 360 x 360. Grid size of FENNS: 18 x 18 x 18, other: see Table 4. Times in ms.*

Radius	Green/Hoetzlein	FENNS
0.2	450^3	18^3
0.3	300^3	18^3
0.5	180^3	18^3
0.8	112^3	18^3
1.0	90^3	18^3

Table 4: *Grid sizes for different neighbor radii.*

$l \in \left[\frac{m}{25}, \ldots, \frac{m}{5}\right]$. Note that we cannot choose larger grid sizes which would exceed the maximum amount of shared memory. On our evaluation GPUs, this size is only about $24KB$ depending on the underlying *compute capability* [NVI19], leading to a maximum of $48KB$ of additional global memory. In comparison to our method, the two other approaches use up to several GBs of additional memory leading to lots of scattered reads.

5. Conclusion

We have shown that our algorithm significantly benefits from recently introduced hardware features like shared memory in contrast to other methods. We use a two-step approach that leverages a single uniform grid in global memory in the first place. Afterwards, we subdivide the previous cells into a smaller grid without the need for additional global memory.

From a theoretical point of view, the overall memory-complexity of $O(m^3)$ remains the same. In practice, m is dramatically smaller compared to other methods since we require a coarse-grained uniform grid in global memory only. Our processing allows us to subdivide this grid into considerably smaller grids that are instantiated on-the-fly in shared memory. This leads to an overall memory consumption of several kilobytes in practice in comparison to several gigabytes of additional memory. Unfortunately, our approach is limited by the shared-memory size. An increase of the grid resolution can lead to better run-time performance since less particles will be assigned to the same grid cell. However, this results in a larger grid that has to be stored in shared memory. We do not consider this as a major limitation since the amount of this fast on-chip memory has grown continuously in the last years.

Even though, we do not require lots of memory, we are also able to improve the overall run-time performance by a factor of up to 80 (ignoring edge cases). If the lookup radius is too large (too many particles fall into our border-regions), we often record slowdowns of up to 3.6 compared to the other approaches on our evaluation scenarios. In practice, the lookup radius is typically chosen in such a way that every particle has a certain number of neighbors on average which implicitly avoids large radii.

In the future, we want to experiment with device-wide synchronization primitives to eliminate the remaining amount of required global memory. We could perform the first step of our algorithm with shared memory only. Furthermore, we want to extent our two-step method with additional steps that could be beneficial in the case of very large scenarios.

Acknowledgements

The authors would like to thank Wladimir Panfilenko, Thomas Schmeyer and Gian-Luca Kiefer for their suggestions and feedback on the paper.

References

[Ben75] BENTLEY J.: Multidimensional Binary Search Trees Used for Associative Searching , 1975. 2

[BP11] BURTSCHER M., PINGALI K.: An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm, 2011. 2

[BVAT17] BRITO C., VIEIRAESILVA A., ALMEIDA M., TEIXEIRA J.: Large Viscoelastic Fluid Simulation on GPU. In *Proceedings of SBGames* (2017). 2

[Fle09] FLEISZ M.: Photon Mapping on the GPU, 2009. 2

[GHOI14] GIESEKE F., HEINERMANN J., OANCEA C., IGEL C.: Buffer kd-Trees: Processing Massive Nearest Neighbor Queries on GPUs. In *Proceedings of the 31 st International Conference on Machine Learning* (2014). 2

[Gor08] GORADIA R.: GPU-based Adaptive Octree Construction Algorithms, 2008. 2

[Gre08] GREEN S.: Particle-based Fluid Simulation. In *Game Developers Conference* (2008). 2

[Gre12] GREEN S.: Particle Simulation using CUDA, 2012. 2

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive SPH Simulation and Rendering on the GPU. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation* (2010). 1

[Gup12] GUPTE S.: Real-Time Photon Mapping on GPU. *Parallel Computing* (2012). 2

[Hoe13] HOETZLEIN R.: Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids, 2013. 2

[KE10] KROG O., ELSTER A.: Fast GPU-based Fluid Simulations Using SPH, 2010. 2

[KGIN11] KAWADA N., GAN B., IMRAN I., NINOMIYA H.: Bounding Grid Algorithm for Calculating Particle Interactions in SPH Simulations. In *Procedia Engineering 14* (2011). 2

[KK16] KOESTER M., KRUEGER A.: Adaptive Position-Based Fluids. *International Journal of Computer Graphics & Animation (IJCGA)* (2016). 1

[KK18] KOESTER M., KRUEGER A.: Screen Space Particle Selection. In *Eurographics Proceedings 2018* (2018). 2

[KSC*14] KOFLER K., STEINHAUSER D., COSENZA B., GRASSO I., SCHINDLER S., FAHRINGER T.: Kd-tree Based N-Body Simulations with Volume-Mass Heuristic on the GPU. *IEEE International Parallel and Distributed Processing Symposium Workshops* (2014). 2

[KSG15] KOESTER M., SCHMITZ M., GEHRING S.: Gravity Games - A Framework for Interactive Space Physics on Media Facades. In *Proceedings of the International Symposium on Pervasive Displays* (2015), ACM. 2

[MG16] MERRILL D., GARLAND M.: Single-pass Parallel Prefix Scan with Decoupled Look-back, 2016. 2, 3

[MHHR07] MUELLER M., HEIDELBERGER B., HENNIX M., RATCLIFF J.: Position Based Dynamics. *3rd Workshop in Virtual Reality Interactions and Physical Simulation VRIPHYS* (2007). 1

[MM13] MACKLIN M., MUELLER M.: Position Based Fluids. *Siggraph2013* (2013). 1

[NVI14] NVIDIA: *Faster Parallel Reductions on Kepler*, 2014. 6

[NVI19] NVIDIA: *CUDA C Programming Guide v10*, 2019. 3, 8

[Ota13] OTAIR M.: Approximate k-Nearest Neighbour Based Spatial Clustering usind kd-Tree. *International Journal of Database Management Systems (IJDMS)* (2013). 2

[Ped13] PEDERSEN S.: Progressive Photon Mapping on GPUs, 2013. 2

[QMN09] QUI D., MAY S., NUECHTER A.: GPU-Accelerated Nearest Neighbor Search for 3D Registration, 2009. 2

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In *23rd IEEE International Parallel and Distributed Processing Symposium* (2009). 2

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-Time KD-Tree Construction on Graphics Hardware, 2008. 2