EG UK Computer Graphics & Visual Computing (2020)
P. D. Ritsos and K. Xu (Editors)

CLAWS : Computational Load Balancing for Accelerated Neighbor Processing on GPUs using Warp Scheduling

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

[1]DFKI Saarbrücken, Germany

Abstract

Nearest neighbor search algorithms on GPUs have been improving for years. Starting with tree-based approaches in the middle 70's, state-of-the-art methods use hash-based or grid-based methods. Leveraging high-performance hardware functionality decreases runtime of these search algorithms. Furthermore, memory consumption has been decreased significantly as well using Shared Memory. In the scope of these enhancements, particles have been reordered by different constraints that simplify neighbor processing. However, inspecting the existing algorithms reveals underused capabilities caused by algorithm desing. Exploiting these capabilities in a smart way can increase occupancy and efficiency on GPUs. In this paper, we present a neighbor processing approach that is based on dynamic load balancing. We rely on a lightweight workload-analysis phase that is applied during neighbor processing to distribute work throughout all warps in a thread group on-the-fly. In different domains, the neighbor function is often symmetric and, thus, commutative in each argument. In contrast to prior work, we use this domain knowledge to reduce the number of memory accesses considerably. Measurements of the newly introduced features on our evaluation scenarios show a comparable runtime performance to state-of-the-art methods. Increasing the overall workload by processing million-particle domains leads to significant improvements in terms of runtime. At the same time, we minimize global memory consumption to enable more particles to be processed compared to current approaches.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ *Shared memory algorithms; Massively parallel algorithms; Graphics processors;*

1. Introduction

Nearest neighbor search is an indispensable feature in different methods, for example in *Position Based Dynamics* [MHHR07] or in *Position Based Fluids* [MM13, KK16]. To simulate real-world behavior, these approaches perform multiple iterations to determine forces between and apply them to particles. To compute these forces, a detection of neighbor particles has to be done in a short time range of several milliseconds. This ensures a fluent creation of updated particle datasets rendered in real-time without having visual effects like lags [MMCK14, KSG15, KSZD15, KK18]. The neighbor determination concept is also needed in other areas in computer graphics, for instance *photon mapping* [GPGSK18]. Photon particles are shot into a scene and collected afterwards to calculate the degree of illumination in a certain range around a specified point.

In all of these methods, the amount of particles is usually limited to ensure fast processing and achieve real-time performance. However, it can be necessary to increase the amount of particles significantly in other kinds of simulations. Regarding an agent-based model [Kos18], it can be necessary to have a large amount of agents to simulate realistic scenarios. If we want to check a given behavior within a metropolitan area, we can easily reach millions of agents that have to be considered. In this case, the runtime does not have to be real-time, since the result is not as time-critical as in the presented computer-graphics scenarios. Instead, the simulation should deliver a suitable result in a reasonable time with respect to the amount of agents [Mil16]. For interaction between agents, a nearest neighbor step has to be performed to find a corresponding interaction agent. This search step is similar to the ones in particle-simulation domains.

Nearest neighbor search has been investigated for many years primarily in the scope of tree-based methods. Especially, the k-d-tree construction has been heavily explored [ZHWG08] and there have been many improvements in terms of algorithms as shown in [Ota13, GHOI14]. After the appearance of fast grid-based searches [Gre08], runtime could be reduced significantly. However, there are still opportunities for further optimization and increase of performance leveraging novel hardware-based features.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

In this paper, we provide concepts to increase occupancy of a GPU-based neighbor search and its associated processing step. Our work is based on the method and its algorithms by Groß et al. [GKK19]. In the first step, we introduce a commutative particle processing scheme to reduce the number of memory accesses. In the second step, particles are reordered in a different way to form larger contigous chunks in memory that enables us to realize fused neighboring-cell lookup tables. This reduces the number of random memory accesses by cutting down the number of entries in these tables. Finally, we present a novel warp-scheduling algorithm that averages the work per warp in each thread group. This significantly helps to increase the occupancy on the whole device level. After a comparison to related work, the novel algorithms are presented in Section 3. In Section 4 the performance is compared to the state-of-the-art methods. In the end, we give a short summary and an outlook about upcoming topics.

2. Related Work

Regarding nearest neighbor search, grid-based methods are the state-of-the-art approaches to ensure fast performance. Grid cells are predestined for being processed efficiently in parallel, since their simple memory layout is most suitable for GPUs. Green presented two algorithms using a grid to improve nearest neighbor search [Gre12]. In his first version, he uses mappings to store the relation between a grid cell index and a particle index. These mappings are sorted by grid cell index using a *Radixsort* [SJ17]. In a second version, he enhances this approach using novel low-level hardware functionalities like *Atomic* functions to count the particles in each cell [NVI19]. Hoetzlein improves this approach [Hoe13] by avoiding slow *Radixsort* passes. In order to create a histogram of particles, he also uses *Atomic* functions. Instead of using a sorting algorithm, he uses a *prefix scan* algorithm [MG16] to calculate the start indices per grid cell and reorder particles accordingly.

Groß et al. improved Hoetzlein's approach by presenting their method *FENNS* [GKK19]. They introduce a concept of differently typed particles that is described in Section 3 in more detail. Furthermore, they decrease memory consumption significantly by reducing the overall grid size. In addition, an acceleration structure in shared memory has been designed in the scope of a group-level algorithm. This increases processing performance, since a grid cell contains many more particles and, hence, they achieve a higher occupancy of the warps. Although there is comparable computational overhead, they realize compatible runtime with respect to other state-of-the-art approaches.

Algorithms that create different workloads in warps can lead to gaps that can be used by other warps. To improve utilization, it is beneficial to use methods based on the concept of dynamic warp scheduling. Warps having a high workload are determined and their work is split into different smaller parts. These parts are processed by the idle warps to increase occupancy and runtime performance. An example of such a warp scheduling technique is given by Narasiman et al. [NSL*11]. They provide a warp-based microarchitecture to analyze branch divergencies within a warp in a two-level approach. Using a reordering, they can increase the occupancy by avoiding idle threads in the scope a warp. This approach

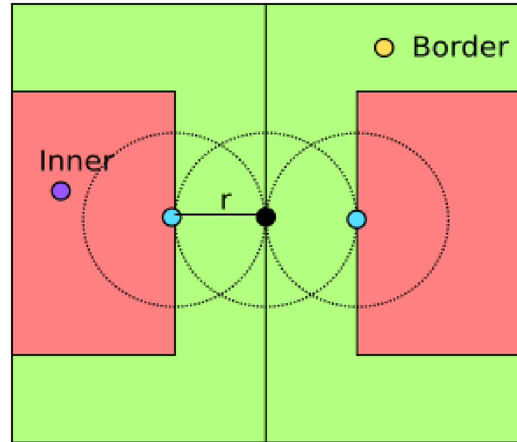

Figure 1: *The relations between the two different regions that contain either inner particles or border particles. The distance between a grid-cell border (in the middle) and a virtual inner grid-cell border (the edges of the red rectangles) is exactly the search radius r. The inner particles (violet, blue) can only reach particles within their own grid cell. Border particles (black, yellow) can reach potential neighbor particles in two or more neighboring grid cells.*

aims to realize an inner-warp scheduling, but does not cover general warp-scheduling tasks.

Another approach to increase performance is *Dynamic parallelism* provided by NVIDIA [NVI14a]. In this method, GPU kernels are launched within another GPU kernel. This enables recursive method calls on the GPU side. Otherwise, the intermediate result has to be checked on the CPU-side and a GPU kernel has to be invoked afterwards. Using this feature reduces the workload on CPU-side and the memory transferring overhead between both devices.

Regarding different workloads, load balancing is another common concept. In general, common approaches estimate a workload distribution and send parts of their work to other processing units, if there is an imbalance between these loads. Especially in parallel units, this concept plays a major role [RSG10, KB16]. A method using a real-world scenario is presented by Sthapit et al. [SHT17]. An adaptive approach using a functional performance model is introduced by Clarke et al. [CLR11]. Utilizing load balancing in the field of GPUs is presented in [PPCS*15]. However, the presented concepts of scheduling warps cannot be applied to our neighbor processing task.

3. Concepts

FENNS introduces a particle-category concept to divide particles into two different groups. Depending on their locations within a grid cell, a particle is either an inner or a border particle. This leads to a splitting of the grid cell into two different areas. The category specification of a particle is given as follows: Since the search radius and the resulting grid are fixed, they are used to determine the areas in each grid cell. Consider a particle located exactly on the edge of a grid cell. Potentially, there are reachable particles with

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

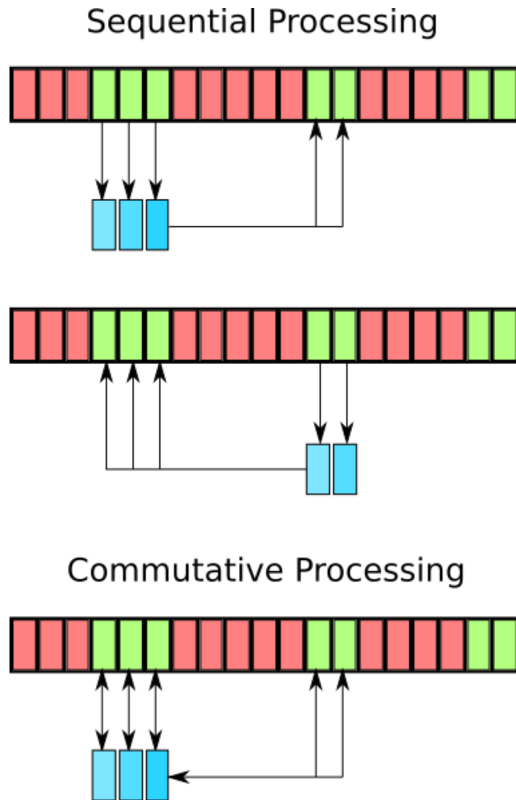

Figure 2: Top: *During sequential neighbor processing, particles are loaded from global memory into shared memory (blue). They process their neighbors and store the results. In a following step, the neighbor particles are loaded and the previous particles have to be accessed again. This leads to a doubled load amount, since every particle has to be accessed in global memory and stored in shared memory.* Bottom: *Using commutative processing, we can reduce the amount of loads, since only one part of the particles has to be moved into shared memory. The processing is done and the results are stored in the respective particle data. The second step of the sequential processing can be skipped in this case.*

respect to the search radius in the neighboring grid cells. Since we can only reach particles up to this radius, we create a virtual border within each grid cell (see Figure 1). Considering a particle located on any point of the border, the result is a virtual grid cell with dimension $gridCellLength - 2 * searchRadius$. Every particle within this virtual grid cell is an inner particle. This kind of particles can only reach neighbors in their own grid cell during neighbor processing. Particles outside of the virtual grid may have potential neighbor particles in neighboring grid cells and are called border particles. Since the neighbor grid cells are also divided, only border particles from the own and the surrounding cells have to be checked for neighbor relations.

3.1. Commutative Processing

In the previous work by Groß et al., they load particle data (from a given input buffer) into shared memory. They look up particles of the neighboring cells in global memory and use this data to perform calculations like forces. As soon as all loaded particles have been processed with respect to their neighbors, the results are stored and the next chunk of particles is processed. Note, that we have to apply this step also for the already inspected neighbor particles. This means, the neighbor particles are accessed in global memory again. By definition, we know that the previously loaded particles are in range and have to be handled. This leads to a second load from global memory to fetch the previously loaded particle data. Since lookups of particle data always happen in both directions, we can skip one access.

The commutative processing is only possible, if a symmetric neighbor-kernel function is used. A prominent example domain is *Smoothed Particle Hydrodynamics* (SPH), in which such neighbor kernels are used [Mon92]. Smoothing kernels in SPH are usually applied to each neighbor and the other way around [MCG03, SSP07, KK16]. As a consequence, we merge both processing directions into one single step. Using this method, we are able to reduce the amount of loads significantly (see Figure 2).

Considering FENNS, processing of neighbors within a grid cell can be handled in a commutative way. Only particles in the loaded chunk has to be excluded, since they are processed in parallel. A parallel commutative applying would result in a doubled handling. For this reason, we apply the processing in the sequential way to avoid wrong results. To process the remaining neighbors, we have to split the for-loop of *FENNS* into different loops. These loops handle either inner particles or border particles, since they are located in different memory regions (compare Figure 3).

3.2. Particle Reordering

Particle reordering is the first step in nearest neighbor search. This is needed, since particle data has to be processed in blocks. To access these blocks easily, we need the start indices of each block. Moreover, the amount of particles can be derived from these indices. The index buffer is usually generated during the particle sorting step, while all particles are moved to their appropriate memory locations. In *FENNS*, the particles are ordered in a two way approach. First, they are ordered by their underlying grid cell index. Second, the particles are ordered by being either an inner particle or a border particle (see Figure 3, left).

For the following concepts (see Section 3.3 and Section 3.4) a different particle reordering is needed. This time, the ordering steps are switched. In the first step, the particles are sorted by being an inner or border particle, while in the second step the particles are sorted by grid cell index (see Figure 3, right). This layout is more suitable, since memory is accessed in a more efficient way during processing on GPUs. Although the border particle data has to be loaded according to the lookup table, a three-cell chunk can be loaded in one step using coherent memory accesses. Sorting this way ensures that the inner particle section has to be accessed only once. For neighbor processing, only the border-particle section has to be taken into account. A further advantage is that all accesses to border particles can be performed consecutively.

The implementation of this step is done as follows: First, we build a histogram of inner and border particles per grid cell in an

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

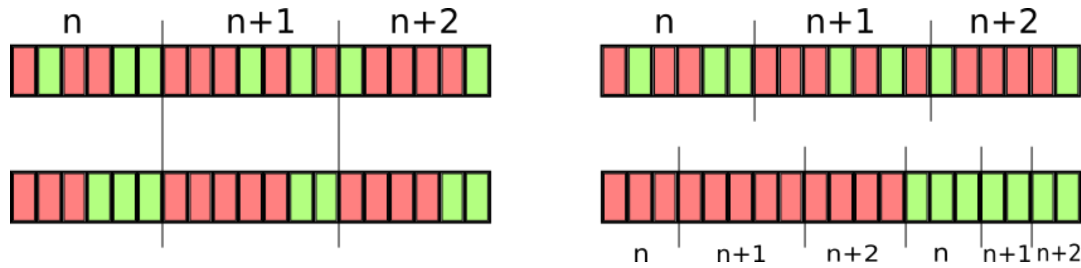

Figure 3: Left: *The sorting of particles in [GKK19]. The particles are sorted by grid cell and the by being an inner(red) or a border particle(green).* Right: *The new sorting result. The particles are first sorted by being an inner or a border particle, then by grid cell. This is advantageous, since in the first processing step all inner particles are processed. In the second step, only border particles has to be processed. The particle data is close to all other border particles. Furthermore, the memory access pattern is improved, since we can leverage coherent memory accesses.*

additional buffer using *AtomicAdds*. All inner particles are in the first part and the border particles in the last part of this buffer (depicted in Figure 3). Second, the buffer is handled completely by an inclusive prefix sum to get the global end indices of each grid-cell subsection. Third, the particles are reordered into their appropriate location by drawing a number from the buffer count using *AtomicAdd* again. This *Atomic* function is used with value -1 to decrease the counter meaning that the buffer is filled from the end to the start.

3.3. Merging Lookup Table Accesses

As a third concept, we can shrink the lookup table from 27 to 9 entries in a 3D-scenario. Since the table always refers to three consecutive grid cells, we can use this fact to reduce the amount of lookups. Instead of inspecting only one grid cell after another and lookup the start and the end index each time, we consider three grid cells as one large virtual grid cell (see Figure 4). In this case, we only have to retrieve the start index of the first cell and the end index of the last cell. In our implementation, we used the lower values as it is shown in the right grid in Figure 4. In essence, the programmer has to decide which values should be stored. However, during neighbor processing, the current grid cell has to be excluded, since calculations within a grid cell are already done. This special case has to be handled in line 24 in Algorithm 1.

3.4. Warp Scheduling

In the implementation of FENNS, the neighbor grid cells are processed in parallel using one warp per cell. This always causes idle warps, since this implementation uses 26 warps, although our target GPU architecture provides 32 in one group. Furthermore, there can be cases related to grid cells in corners or at the edges of a given scene that have empty neighbor grid cells located outside of the scene. The worst case (a corner grid cell) only occupies 7 warps at maximum (see Figure 5, left). The goal of this method is to utilize idle warps and increase occupancy. The workload balance of all warps is calculated in the beginning to check the work distribution. Afterwards, we reduce the amount of work per warp by sharing work in two or more warps. The result is a smoother distribution throughout a group. Especially in million particle domains, this can increase performance, since the work can be distributed and processed in a more efficient way (see Figure 5, right).

In general, the hardware-implemented warp scheduler takes care to occupy all warps on a device. If only 26 warps are needed during the neighbor processing, the remaining warps are assigned to other groups to achieve occupancy. However, our goal aims to another kind of occupancy that achieves better performance at the same time. Consider the case, where a single warp has to handle a huge amount of particles. This results in a sequential processing. The hardware scheduler assigns other groups to the remaining warps. This leads to scattered memory accesses and hence to performance loss. If we move work from this performance-critical warp to the other warps, we benefit from coherent memory accesses for this particle chunk resulting in a higher performance.

Our warp scheduler is embedded into the border-particle-processing step shown in Algorithm 1. To prepare the information for warp scheduling, we only need to invoke the first warp (Algorithm 1, line 2). The remaining warps wait until the scheduling pass has finished at line 16. Since the lookup table has been reduced as described in Section 3.3, only the first 9 lanes have to load their associated particle histogram into a temporary mapping buffer in shared memory (lines 3 - 13). This histogram is needed to decide the schedule of warps. The scheduler is started afterwards (line 14).

Using the given histogram, the scheduler creates results for three shared memory buffers containing maps and values for further processing. The dependencies and example values are shown in Figure 6. The first buffer *WarpMap* contains the target warp group index and is needed to access the right values in the other two buffers. In Figure 6, the first three warps belong to group 0, warp 4 and 5 to 1 and so on. The second buffer *TargetMap* contains the origin warp index. If only warp 0, 1, 3 and 4 need to work, we can access the values by using *WarpMap* and *TargetMap*. The third buffer *WarpStarts* contains the amount of warps that are scheduled for a specific group (group 0 needs 3 warps, for example). We can use the same lookup stored in *WarpMap* for *WarpStarts* and *TargetMap*. This buffer is only needed to determine the stride length of the warp stride loop and the corresponding starting indices per lane in Algorithm 1 lines 21 and 22.

When the scheduler has finished, we can assign each warp to its calculated group index. This index is needed to lookup the target

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

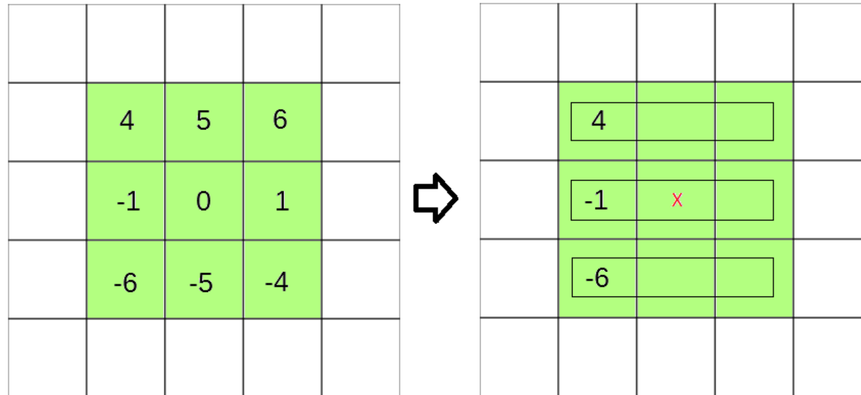

Figure 4: Left:*A 2D-lookup table with relative indices. The relative grid cell is cell 0.* Right: *A reduced lookup table that only contains the beginning of a 3-cell block. Since there are always three cells in a row, we can reduce the lookup table and merge these cells into one cell.*

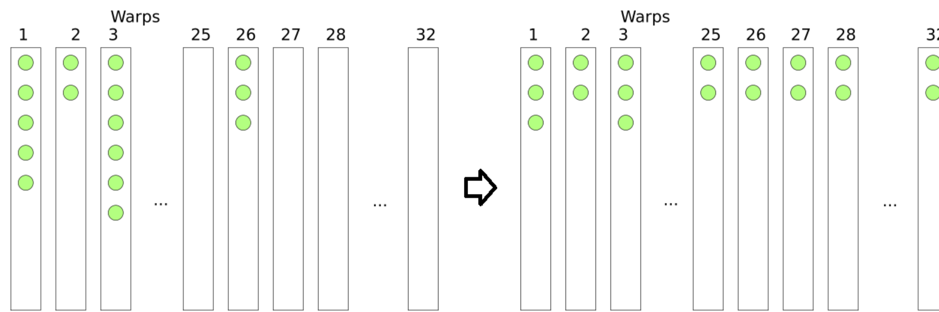

Figure 5: Left: *During neighbor processing, one warp processes the particles of one neighbor cell. Since there are 26 neighbor cells in a 3D-domain, this leads to at least 6 empty warps. Regarding edges or corners of a given scene, there can be further empty warps.* Right: *To increase occupancy, a large amount of particles within a warp can be split and assigned to different warps. This leads to a smoother distribution and uses all available warps in a group.*

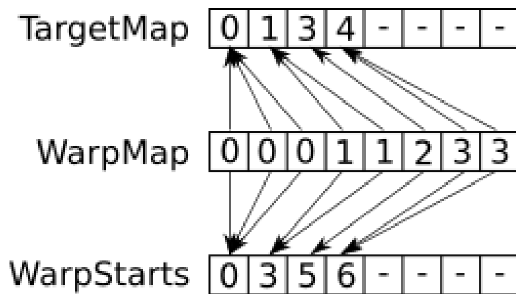

Figure 6: *To increase occupancy and schedule all warps, a warp map is needed. Assuming a warp size of 8 and 4 occupied warps, we get such a warp map for example (middle). This map is used to find the underlying grid cells with help of a target map that contains the grid cell indices (top). For processing using grid-stride loops, they need knowledge about the amount of warps that are involved in processing a specified grid cell. For this reason, we need to store an additional start index within the parent thread-group (bottom).*

warp position according to the mapping in Figure 6 and, thus, the global grid cell index (lines 18 and 19). In the following loop, the neighbor processing takes place. After initializing the values and buffers for processing, the step size is retrieved using the *WarpStarts* buffer mentioned above.

Algorithm 2 presents the warp scheduling process. First, we prepare the *TargetMap* by setting the lane index to each warp that has to process particles. Otherwise, the value is set to *intMax* to sort these values ascendingly in line 9 (using a parallel *bubble sort* algorithm [Mun15]). Second, the scheduling part is started also creating the *WarpStarts* buffer in the course of execution. In line 10, the total amount of particles is calculated using a warp reduction [NVI14b]. A percentage distribution over all warps is provided per warp in the next step. Due to rounding artifacts, the calculated amount of warps is in range between 0 and 64. For this reason, an adjustment has to be applied to reach exactly 32 warps. In the first case (lines 12 - 21), the amount of warps is larger than 32 and therefore the amount needs to be reduced. In this step, we have to take care that warps are not disabled accidentially and we have to check that there is at least one warp enabled (line 15). In the second case (lines 22 - 31), the groups are pumped up to 32 warps to achieve maximum occupancy. Again, we have to take care that disabled warps are not enabled ac-

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

cidentially (line 25). After storing the result (line 33), we calculate the start indices of the warps and create the *WarpStarts* buffer (see Algorithm 3). The reordering step and the following scan SHG08 distributes the warps such that every warp maps to its target warp as shown in Figure 6 (see lines 36 - 38).

The start indices calculation method is shown in Algorithm 3. In the beginning, the *WarpStarts* buffer is used as a temporary buffer to check, if a warp is used or not. Using an exclusive scan provides the new index of this warp in the compacted version. In line 9, the index is used to store the amount of warps per group in the right position. The last scan stores the start indices in the buffer. Algorithm 4 shows the implementation for reordering all warps to their new position. The target position of each warp is stored in a temporary variable. Afterwards, the gap between the starts is calculated (lines 2 - 7) and the map is reset to 0. In the following step, the gap is stored at the determined position retrieved from the temporary variable.

Algorithm 1: Processing particles in surrounding grid cells

```
   Input: counter, globalGrid, globalLookup
 1 initialize values and memory
 2 if warp index == 0 then
 3     if lane index < 9 then
 4         pos := grid index + globalLookup[lane index];
 5         for int i = pos; i < pos; i++ do
 6             if i >= 0 & i < globalGrid.Size then
 7                 warpMap[lane index] +=
 8                     counter[i + 1] - counter[i];
 9             end
10         end
11     else
12         warpMap[lane index] = 0
13     end
14     ScheduleWarps(warpMap, targetMap, warpStarts);
15 end
16 group barrier;
17 targetWarp := warpMap[warp index];
18 targetWarpPos := targetMap[targetWarp];
19 gridPos := grid index + globalLookup[targetWarpPos];
20 for int i = start + group index; i < end; i += group size do
21     initialize neighbor processing and values
22     stepsize := warpStarts[targetWarp] * warp size;
23     for int k = borderStart; k < borderEnd; k += stepsize
        do
24         if k < start | k >= end then
25             LookupNeighborCells();
26         end
27     end
28     store results
29 end
```

3.5. Implementation Details

The presented concepts are implemented in C# using the ILGPU compiler [†]. The shown algorithms are designed to be executed on GPUs. The termination conditions of for-loops are padded to multiples of the group size to avoid thread divergencies and, hence, performance loss.

Algorithm 2: Workload calculation and scheduling

```
   Input: warpMap, targetMap, warpStarts
 1 diff := SharedMemory<int>(1);
 2 if warpMap[lane index] > 0 then
 3     targetMap[lane index] = lane index;
 4 else
 5     targetMap[lane index] = intMax ;
 6 end
 7 warp barrier;
 8 SortAscending(targetMap);
 9 amountWarps := WarpReduce (warpMap[lane index]);
10 percentage := DivRoundUp(
11     warpMap[lane index] * warp size,
12     amountWarps);
13 if amountWarps > warp size then
14     diff[0] = amountWarps - 32;
15     while diff[0] > 0 do
16         if percentage > 1 then
17             if AtomicAdd (diff[0], -1) > 0 then
18                 percentage−−;
19             end
20         end
21     end
22 else
23     diff[0] = 32 - amountWarps;
24     while diff[0] > 0 do
25         if percentage > 0 then
26             if AtomicAdd (diff[0], -1) > 0 then
27                 percentage++;
28             end
29         end
30     end
31 end
32 warp barrier;
33 warpMap[lane index] = percentage;
34 CalculateStartIndices(warpStarts, warpMap);
35 WarpScanInclusive (warpMap);
36 ReorderWarps(warpMap);
37 if lane index > 0 &
38 warpMap[lane index] - warpMap[lane index- 1] != 0 then
39     warpMap[lane index] = 1;
40 else
41     warpMap[lane index] = 0;
42 end
43 WarpScanInclusive (warpMap);
```

† www.ilgpu.net

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

All buffers are known beforehand and can be allocated before running the algorithms. The buffer sizes are also padded to a multiple of the warp size [KLH*14, KGK19, KGK20]. The surplus elements caused by padding had to be excluded from calculations and serve as dummy elements without any effect.

Algorithm 3: Calculation of warp start indices

```
   Input: warpStarts, warpMap
 1 if warpMap[lane index] > 0 then
 2 |   warpStarts[lane index] = 1;
 3 end
 4 warp barrier;
 5 WarpScanExclusive (warpStarts);
 6 newIndex = warpStarts[lane index];
 7 warp barrier;
 8 if warpMap[lane index] > 0 &
 9 newIndex < warp size & newIndex >= 0 then
10 |   warpStarts[newIndex] = warpMap[lane index];
11 end
12 WarpScanExclusive (warpStarts);
```

Algorithm 4: Reordering warps in the scheduling step

```
   Input: warpMap
 1 temp:= warpMap[lane index];
 2 if lane index < warp size −2 then
 3 |   gap :=
 4 |       warpMap[lane index + 1] − warpMap[lane index];
 5 else
 6 |   gap := warp size − warpMap[lane index];
 7 end
 8 warp barrier;
 9 warpMap[lane index] = 0;
10 warp barrier;
11 if gap > 0 then
12 |   warpMap[temp] = gap;
13 end
14 WarpScanInclusive (warpMap);
```

4. Evaluation

4.1. Scenarios

For performance measurements, different scenarios have been created and evaluated. The actual concepts are tested against the underlying implementation of *FENNS*. As additional reference results, the methods of Hoetzlein and Green are also implemented to discuss the differences in terms of runtime. Different amounts between 1 million and 8 million particles are measured in these scenarios. Thereby, the particles are uniformly distributed throughout the scene. The scene dimensions are also changed to create cases using a different amount of potential neighbor particles within a grid cell. The scene dimension varies from 90 to 180 in world coordinates with steps of 30. Furthermore, the search radius is varied affecting the neighbor amount directly. For realistic scenarios, the measurements are setup that a particle has approximately 30-40 neighbors. For this reason, the radius was measured in range of 0.5 to 1.5 in world coordinates and is divided in 0.25 steps. However, since the shared-memory approaches use coarser grids in the first level, the particle amount within a certain grid cell is always larger compared to Hoetzlein's and Green's approach. Given a specified particle, the average amount of neighbors is calculated by $\frac{particles}{gridsize}$, where gridsize is $\left(\frac{scenesize}{2 \cdot radius}\right)^3$.

4.2. Runtime

For runtime measurements, we used two NVIDIA devices (GeForce GTX 980 Ti and GeForce GTX 1080 Ti) to check the behavior using different kinds of compute capabilty. The measurements show that the 980 Ti usually needs twice as long as the 1080 Ti. All scenarios are executed in 100 runs and the result tables show the median and the standard deviation. The time is measured in milliseconds[ms].

Table 1 shows the runtime in different scenarios using a fixed average amount of particles. Using the equation mentioned above leads to an average neighbor amount of 38.84 in each scenario. The measurements show that Green's approach is always slower than Hoetzlein's and for this reason, the results are omitted. In these scenarios, *CLAWS* is always slower than Hoetzlein's in a range from approximately 3 to 6 ms. These slowdown results from the additonal overhead in our approach, since the neighbor search is more sophisticated and we have to check a larger amount of potential neighbors due to the coarse grid. However, except in the 1 million case, we are also significantly faster than *FENNS*. In this case, the slowdown is only up to 3.2%, while the speed up in the other cases is up to 75.1%.

Moreover, the slowdowns are not that problematic, since *CLAWS* aims to a million particle domain. Especially the case, in which the amount of particles per grid cell increases, it shows its advantage in terms of runtime. This increasement occurs, if we consider regions with a high density of particles. This can be the case, if we use compressible fluids for example. A closer look on Table 2 shows the significant speed up, if we consider millions of particles. As baseline, we use the scenario with following parameters: The scene size is 90, the amount of particles is 1 million and the search radius is set 1.5. Afterwards, we double the amount each time and measure runtime. From the case having an average of 77.67 neighbors, our approach is fastest with respect to the other methods. This effect becomes more obvious, if the average is doubled again. Using 8 million particles, we can process neighbor up to 2.5x faster.

4.3. Memory Consumption

The complexity of the memory consumption in all presented methods is m^3, where m is the grid size in one dimension [GKK19]. Although complexity is not changed, the real memory consumption has been decreased significantly between Green's/Hoetzlein's and Groß's approach. Green and Hoetzlein use large grids and have therefore a high memory consumption up to several GB. Groß only needs a small grid that fits in shared memory and this fact limits the consumption to several KB of memory. Since the applied improvements are based on the latter one, we can still benefit from a low memory usage. Only the consumption in shared memory is

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

SceneSize	NumParticles	Radius	Algorithm	980 Ti	σ	1080 Ti	σ
60	1048576	1.0	Hoetzlein	15.35	0.64	7.92	0.38
*	*	*	FENNS	21.37	1.12	10.72	0.06
*	*	*	CLAWS	22.64	1.24	11.07	0.68
*	8388608	0.5	Hoetzlein	127.16	5.31	76.46	2.02
*	*	*	FENNS	242.76	1.77	137.24	27.14
*	*	*	CLAWS	155.63	0.78	82.94	2.28
90	1048576	1.5	Hoetzlein	14.66	0.65	7.75	0.0
*	*	*	FENNS	20.63	0.88	11.55	0.02
*	*	*	CLAWS	22.58	0.99	11.67	0.03
*	8388608	0.75	Hoetzlein	139.09	34.09	71.18	4.09
*	*	*	FENNS	239.17	122.43	125.79	51.7
*	*	*	CLAWS	147.8	2.08	77.14	1.21
120	*	1.0	Hoetzlein	134.36	50.29	68.4	6.97
*	*	*	FENNS	236.27	107.72	126.71	46.07
*	*	*	CLAWS	145.57	1.32	75.96	1.58
150	*	1.25	Hoetzlein	132.02	48.17	67.66	6.83
*	*	*	FENNS	230.6	72.13	123.2	46.0
*	*	*	CLAWS	138.86	1.09	72.3	1.43
180	*	1.5	Hoetzlein	129.94	45.0	66.81	4.26
*	*	*	FENNS	224.89	76.8	120.9	43.35
*	*	*	CLAWS	132.39	1.19	69.01	1.32

Table 1: *Performance measurements of the evaluation scenarios. Avg. Neighbors is 38.84 in all cases.*

NumParticles	Avg. Neighbors	Algorithm	980 Ti	σ	1080 Ti	σ
2097152	77.67	Hoetzlein	61.25	0.76	28.24	0.91
*	*	FENNS	46.66	7.22	25.18	0.49
*	*	CLAWS	40.27	0.97	23.98	0.96
4194304	155.34	Hoetzlein	232.6	0.38	115.49	1.02
*	*	FENNS	120.43	29.73	66.16	15.78
*	*	CLAWS	93.84	1.02	49.61	2.02
8388608	310.69	Hoetzlein	903.5	949.94	445.24	13.52
*	*	FENNS	524.43	146.65	269.43	17.51
*	*	CLAWS	343.73	2.89	174.02	4.57

Table 2: *Performance measurements of the evaluation scenarios. SceneSize is set to 90, Radius to 1.5.*

slightly higher than in *FENNS*. The shared memory consumption is increased by $3 * warpsize * 4 + 4$ bytes. However, this is not a limitation, since shared memory is only a temporary memory buffer and although it is not that large, it is still not exceeded by this approach.

5. Conclusion

In this paper, we introduced *CLAWS*, a concept to utilize all warps within a nearest neighbor processing step. For this reason, an algorithm to calculate the workload balance has been designed to create a schedule and distribute work accordingly. To improve memory accesses, particles are reordered in an appropriate way in global memory. This ensures faster coherent memory loads. Using a symmetric processing concept, we are able to reduce the amount of loads and stores which are no longer needed.

We showed that these improvements can lead to a significant speed up compared to the existing methods. Especially in million-particle domains and concerning lots of neighbors in a certain place (like compressible fluids), this method can be useful to speed up neighbor processing. Furthermore, we can benefit from a low memory consumption to save resources.

For future work, we can explore the algorithm behavior, if we increase the workload within the neighbor kernels. Moreover, we can evaluate this method using agent-based scenarios.

Acknowledgement

The authors would like to thank Thomas Schmeyer and Alexander Bosch for their suggestions and feedback regarding our method and pre-reviewing the paper.

This work was supported by the SINTEG-project DESIGNETZ funded by the German Federal Ministry of Economic Affairs and Energy (BMWi) under the grant 03SIN222.

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.

References

[CLR11] Clarke D., Lastovetsky A., Rychkov V.: Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity. vol. 6586, pp. 41–50. 2

[GHOI14] Gieseke F., Heinermann J., Oancea C., Igel C.: Buffer k-d trees: Processing massive nearest neighbor queries on gpus. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32* (2014), ICML'14. 1

[GKK19] Gross J., Köster M., Krüger A.: Fast and efficient nearest neighbor search for particle simulations. In *Computer Graphics and Visual Computing (CGVC)* (2019), The Eurographics Association. 2, 4, 7

[GPGSK18] Grittmann P., Pérard-Gayot A., Slusallek P., Krivanek J.: Efficient caustic rendering with lightweight photon mapping. *Computer Graphics Forum* (2018). 1

[Gre08] Green S.: Particle-based fluid simulation. In *Game Developers Conference* (2008). 1

[Gre12] Green S.: Particle simulation using cuda, 2012. 2

[Hoe13] Hoetzlein R.: Fast fixed-radius nearest neighbors: Interactive million-particle fluids, 2013. 2

[KB16] Kupriyashin M. A., Borzunov G. I.: Computational load balancing algorithm for parallel knapsack packing tree traversal. *Procedia Computer Science 88* (2016), 330–335. 2

[KGK19] Köster M., Gross J., Krüger A.: Fang: Fast and efficient successor-state generation for heuristic optimization on gpus. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)* (2019), Springer. 7

[KGK20] Köster M., Gross J., Krüger A.: Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction. *International Journal of Parallel Programming* (06 2020). 7

[KK16] Köster M., Krüger A.: Adaptive position-based fluids: Improving performance of fluid simulations for real-time applications. *International Journal of Computer Graphics and Animation 6* (06 2016), 01–16. 1, 3

[KK18] Köster M., Krüger A.: Screen space particle selection. In *Proceedings of the Conference on Computer Graphics and Visual Computing* (2018), Eurographics Association, p. 61–69. 1

[KLH*14] Köster M., Leissa R., Hack S., Membarth R., Slusallek P.: Code refinement of stencil codes. *Parallel Processing Letters 24* (09 2014), 1441003. 7

[Kos18] Kosiachenko E.: Efficient gpu parallelization of the agent-based models using mass cuda library. 1

[KSG15] Köster 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). 1

[KSZD15] Köster M., Schmitz M., Zehle S., Detzler B.: Asterodrome: Force-of-Gravity Simulations in an Interactive Media Theater. In *14th International Conference on Entertainment Computing (ICEC)* (2015), vol. LNCS-9353 of *Entertainment Computing - ICEC 2015*, Springer International Publishing, pp. 359–366. 1

[MCG03] Müller M., Charypar D., Gross M.: Particle-based Fluid Simulation for Interactive Applications. In *Symposium on Computer Animation* (2003), Eurographics Association. 3

[MG16] Merrill D., Garland M.: Single-pass parallel prefix scan with decoupled lookback. 2

[MHHR07] Müller M., Heidelberger B., Hennix M., Ratcliff J.: Position based dynamics. *J. Vis. Comun. Image Represent. 18*, 2 (Apr. 2007), 109–118. 1

[Mil16] Milla A.: *Crowd Modeling and Simulation on High Performances Architectures*. PhD thesis, 2016. 1

[MM13] Macklin M., Müller M.: Position based fluids. *ACM Transactions on Graphics 32* (07 2013), 104:1–. 1

[MMCK14] Macklin M., Müller M., Chentanez N., Kim T.-Y.: Unified Particle Physics for Real-time Applications. *ACM Trans. Graph.* (2014). 1

[Mon92] Monaghan J. J.: Smoothed particle hydrodynamics. In *Annual Review of Astronomy and Astrophysics* (1992). 3

[Mun15] Mundra M. J.: Minimizing execution time of bubble sort algorithm. 5

[NSL*11] Narasiman V., Shebanow M., Lee C. J., Miftakhutdinov R., Mutlu O., Patt Y. N.: Improving gpu performance via large warps and two-level warp scheduling. Association for Computing Machinery. 2

[NVI14a] NVIDIA: *Dynamic Parallelism*, 2014. 2

[NVI14b] NVIDIA: *Faster Parallel Reductions on Kepler*, 2014. 5

[NVI19] NVIDIA: *CUDA C Programming Guide v10*, 2019. 2

[Ota13] Otair M.: Approximate k-nearest neighbour based spatial clustering using k-d tree. *International Journal of Database Management Systems* (2013). 1

[PPCS*15] Passerat-Palmbach J., Caux J., Siregar P., Mazel C., Hill D.: Warp-level parallelism: Enabling multiple replications in parallel on gpu. *ESM 2011 - 2011 European Simulation and Modelling Conference: Modelling and Simulation 2011* (2015). 2

[RSG10] Rajavel R., Somasundaram T. S., Govindarajan K.: Dynamic load balancer algorithm for the computational grid environment. In *Information and Communication Technologies* (2010), Das V. V., Vijaykumar R., (Eds.), Springer Berlin Heidelberg, pp. 223–227. 2

[SHG08] Sengupta S., Harris M., Garland M.: *Efficient parallel scan algorithms for GPUs*, 2008. 6

[SHT17] Stapit S., Hopgood J., Thompson J.: Distributed computational load balancing for real-time applications. 2

[SJ17] Stehle E., Jacobsen H.-A.: A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, Association for Computing Machinery. 2

[SSP07] Solenthaler B., Schläfli J., Pajarola R.: A unified particle model for fluid-solid interactions. *Journal of Visualization and Computer Animation* (2007). 3

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.* (2008). 1

© 2020 The Author(s)
Eurographics Proceedings © 2020 The Eurographics Association.