Advanced Load Balancing for SPH Simulations on Multi-GPU Architectures

Kevin Verma*† Kamil Szewc* Robert Wille†
*ESS Engineering Software Steyr GmbH, Austria
†Institute for Integrated Circuits, Johannes Kepler University Linz, Austria
Email: {kevin.verma, kamil.szewc}@essteyr.com robert.wille@jku.at

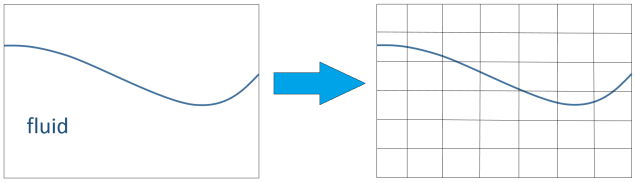
***Abstract*—*Smoothed Particle Hydrodynamics* (SPH) is a numerical method for fluid flow modeling, in which the fluid is discretized by a set of particles. SPH allows to model complex scenarios, which are difficult or costly to measure in the real world. This method has several advantages compared to other approaches, but suffers from a huge numerical complexity. In order to simulate real life phenomena, up to several hundred millions of particles have to be considered. Hence, HPC methods need to be leveraged to make SPH applicable for industrial applications. Distributing the respective computations among different GPUs to exploit massive parallelism is thereby particularly suited. However, certain characteristics of SPH make it a non-trivial task to properly distribute the respective workload. In this work, we present a load balancing method for a CUDA-based industrial SPH implementation on multi-GPU architectures. To that end, dedicated memory handling schemes are introduced, which reduce the synchronization overhead. Experimental evaluations confirm the scalability and efficiency of the proposed methods.**

I. Introduction

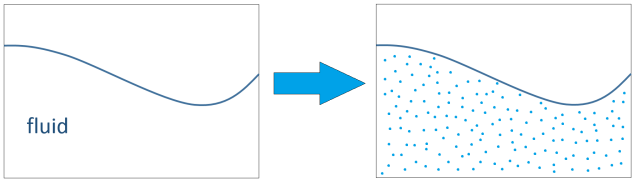
Efficient simulation of fluid-flows is an important engineering challenge today. Corresponding solutions find applications in areas such as geo-engineering, chemical processing, or the automotive industry (for an overview see [1]). In the latter, a proper modeling of such flows is relevant e.g. when designing new car bodies for water management (rain, painting process), developing new tire treads (better friction, avoiding aquaplaning), designing new engine parts, and many other (for an overview of applications in the automotive industry see [2]). The main advantage of numerical simulations is the capability to model complex scenarios which are difficult or costly to measure in the real world.

In general, there exist two main approaches for fluid modeling:

- The most common approach is based on grid-based methods, as sketched in Fig. 1a. Here, the fluid is composed of fluid cells, aligned in a regular grid, each of which contains some given volume of fluid. The grid is fixed in space and usually does not move or deform with time. The material or fluid moves across these fixed grid cells, while the fluid itself is constrained to stay with the grid.
- The second approach to model fluids relies on particle-based methods. The general scheme of these methods is sketched in Fig. 1b. The fluid is discretized by sample points, which are identified as particles. These particles completely define the fluid, which means that the particles move with the fluid. Some of the most common methods belonging to the particle-based method family are: *Moving Particle Semi-implicit* (MPS, [3]), *Finite Volume Particle Method* (FVPM, [4]) or, *Smoothed Particle Hydrodynamics* (SPH, [5]). All of them are relatively new and all these methods are closely related.



(a) Grid-based fluid structure.



(b) Particle-based fluid structure.

Fig. 1: Basic approaches for fluid modeling.

In general, particle-based methods have several advantages over grid-based methods, for details see [6].

In this work, we focus on the SPH technique which received significant interest because the mesh generation can be omitted and the method is suitable to model flows involving complex physics. Because of this, corresponding SPH simulations also got established in industrial practice (see e.g. [7]).

But despite these benefits, SPH suffers from a huge numerical complexity. In order to simulate real life phenomena in appropriate resolution, up to several hundred millions of particles are necessary. Thus, simulations of short periods of physical time frequently require large execution times. Because of this, *High Performance Computing* (HPC) methods need to be leveraged to make SPH techniques applicable for more practically relevant and industrial applications. In this context, the use of methods for *General Purpose Computation on Graphics Processing Units* (GPGPU) is particularly suited since the discrete particle formulation of SPH allows for independent computations, which can be distributed among different GPUs and executed in a massively parallel fashion.

However, certain characteristics of the SPH method make it a non-trivial task to properly split the workload in a way which allows each GPU to complete the respectively assigned computations in the same time. In fact, most approaches proposed thus far suffer from overheads introduced by multi-GPU architectures. Similar approaches are introduced for related methods, which however can not be employed to the SPH method due to different characteristics in terms of particle motion (this is discussed in more detail later in Section III).

In this work, we propose an advanced load balancing scheme for SPH simulations which addresses these shortcomings. The proposed scheme frequently conducts a so-called *domain decomposition correction*, which adjusts the current distribution of the workloads among the GPUs. Since this frequently requires re-allocations of memory, dedicated memory handling schemes are introduced. Experimental evaluations with an industrial SPH solver confirm that sophisticated memory handling allows to reduce the synchronization overhead and, therefore, results in an increased performance.

The remainder of this paper is structured as follows: The next section provides the background on SPH simulations as well as their execution on multi-GPU architectures. Afterwards, Section III discusses the resulting problems which are addressed by the proposed advanced load balancing scheme described in Section IV. Finally, Section V summarizes the obtained results from our experimental evaluations before the paper is concluded in Section VI.

II. Background

In order to keep this work self-contained, this section briefly reviews the basics on the SPH technique as well as the state-of-the-art on multi-GPU architectures used for this purpose.

A. Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a particle-based, fully Lagrangian method for fluid-flow modelling and simulation. This method was independently proposed by Gingold and Monaghan [5] to simulate astrophysical phenomena at the hydrodynamic level (compressible flow). Nowadays, the SPH approach is increasingly used for simulating hydro-engineering applications – involving free-surface flows where the natural treatment of evolving interfaces makes it an enticing approach.

The main ideas of the *SPH* method rely on the following basis: Let J be the set of all considered discrete particles. Then, a scalar quantity A is interpolated at position r by a weighted sum of contributions from J, i.e.

$$\langle A(r) \rangle = \sum\nolimits_{j \in J} A_j V_j W(r - r_j, 2h), \tag{1}$$

where V_j is the volume of the respective particle j, r_j is the position of this particle, and A_j the field quantity at position r_j. W is a smoothing kernel with the so-called smoothing length $2h$ as a width – defining that only particles within a distance shorter than $2h$ will interact with a particle j. This kernel function W is a central part of SPH simulations and the appropriate choice of a smoothing kernel for a specific problem is of great importance. At the same time, a kernel must satisfy three conditions/properties, namely

1) the *normalization condition*

$$\int_r W(r - r_j, 2h) dr = 1 \tag{2}$$

stating that the integral over its full domain is unity,

2) the *Delta function property*

$$\lim_{h \to 0} W(r - r_j, 2h) = \delta(r - r_j) \tag{3}$$

stating that, if the smoothing length $2h$ approaches zero, a delta distribution is applied (with δ being the Dirac delta function), and

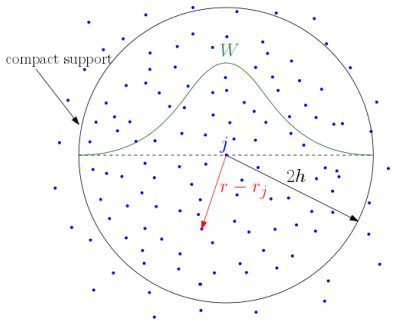


Fig. 2: Illustration of a 2D domain for a particle $j \in J$.

3) the *compact support condition*

$$W = 0 \text{ when } |r - r_j| > 2h \tag{4}$$

ensuring that only particles within the smoothing length $2h$ are considered.

Example 1. *Fig. 2 illustrates a 2D domain with a kernel function W and smoothing length $2h$ for a particle $j \in J$.*

For a more detailed treatment of SPH, we refer to [8], [9].

B. SPH Simulations on Multi-GPU Architectures

The discrete particle formulation of physical quantities makes SPH suitable for parallel architectures. In particular, the sheer number of independent per-particle computations makes SPH a promising method for the *General Purpose Computations on Graphics Processing Units* (GPGPU) technology. Corresponding solutions utilizing the computational power of GPUs have initially been introduced by Kolb and Cuntz [10] as well as Harada et al. [11], where the *Open Graphics Library* (OpenGL) was employed. Later, SPH implementations based on the *Compute Unified Device Architecture* (CUDA) have been developed [12].

However, in order to simulate huge domains involving millions of particles, a single GPU device is usually not sufficient anymore. In these cases, the underlying SPH implementation needs to be distributed over several devices – yielding a *multi-GPU architecture* as originally presented by Dominguez et al. [13]. Here, CUDA and *Message Passing Interfaces* (MPIs) have been employed to parallelize the SPH simulation with up to 128 GPUs where each GPU covered the simulation of up to 8 million particles. Besides that, a similar architecture has also been utilized in the solution proposed in [14].

The performance of both solutions strongly depends thereby on the fact that, in any SPH simulation, neighboring particles need to be frequently accessed during one computational iteration. More precisely, as defined by Eq. 1, the scalar quantity A is interpolated by a weighted function of all particles which are located within the influence radius $2h$ (defined by the smoothing length, as reviewed in Section II-A). Applying a straight-forward nearest neighbor search algorithm, this requires the iteration through the entire fluid domain – yielding a complexity of $O(|J|^2)$. Although polynomial, the sheer number $|J|$ of particles makes this straight-forward approach infeasible for many practically relevant problems (e.g. in applications such as simulations of wave interactions with an off-shore oil rig platform, more than one billion particles have to be considered [7]).

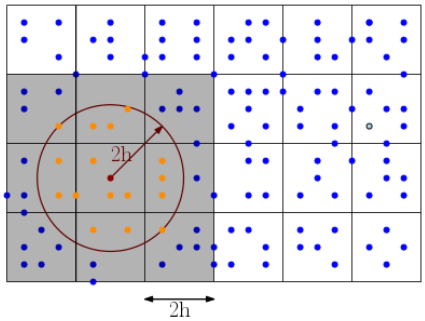


Fig. 3: Virtual search grid.

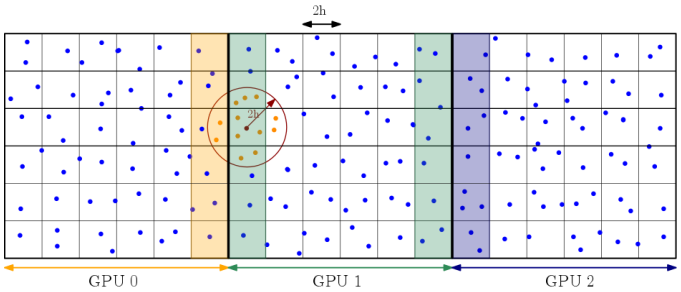


Fig. 4: Virtual search grid distributed to three GPUs.

Hence, corresponding optimizations have been introduced which rely on a so-called *virtual search grid* as illustrated in Fig. 3. Here, the entire fluid domain is divided into a search grid where each cell has the size of the influence radius $2h$. By this, it can be guaranteed that, for a considered particle $j \in J$ (exemplarily denoted by a red dot in Fig. 3), all neighboring particles must be located within the adjacent cells. This way, instead of iterating through the entire fluid domain, only a subset of it (highlighted grey in Fig. 3) has to be considered in order to determine the neighboring particles (denoted by orange dots in Fig. 3).

This search grid not only allows for a fast nearest neighbor search, but also provides a scheme how to divide the entire fluid domain into sub-domains. For example, a spatial subdivision based on the grid cells could be applied – leaving every sub-domain with an equal amount of cells of the grid. These sub-domains can then be distributed to the corresponding devices on a multi-GPU architecture. However, such a primitive approach will not result in an optimal balance of the workload. In fact, there are a few characteristics of SPH simulations which make determining an optimal subdivision of the domain and, hence, load balancing a non-trivial task. This provides the motivation of this work which is discussed in more detail next.

III. Motivation

In general, the goal for every application executed on a multi-GPU architecture is to split the workload in a way which allows each GPU to complete the respectively assigned computations in the same amount of time. Since such a behavior is usually difficult to guarantee, the alternate goal is to minimize the gap between the longest and shortest time consumption.

In the case of SPH simulations, the key to achieve optimal performance is to determine the optimal positions for a subdivsion of the domain. However, for that purpose, important characteristics of SPH need to be considered.

First, although the partitioning of the entire fluid domain into sub-domains (defined by the grid cells) provides a scheme how to distribute the corresponding workload over the respective GPU devices, it is not always useful to apply a spatial subdivision based on the geometry of the domain only. In fact, particles usually do not equally distribute along the domain – making a subdivision based on the geometry unbalanced.

Example 2. *Consider a scenario, where a dam break has to be simulated using SPH techniques. At time step $t = 0$, all particles are clustered at one side of the domain, namely the water reservoir. At the end of the simulation (after the dam broke), most particles are likely been distributed equally among the domain.*

Therefore, a subdivision according to the distribution of particles is essential for most applications.

Besides that, the concept of a virtual search grid for fast nearest neighbor search also needs additional consideration on a multi-GPU architecture. As previously discussed, the computation of a field quantity A_j at a position r requires frequent access to all particles within $2h$. However, on a multi-GPU architecture, the domain needs to be split into sub-domains – each of which is assigned to one GPU device. This yields an architecture where each sub-domain has two neighboring sub-domains, except for those at the perimeter of the domain, which have only one neighbor.

Example 3. *Consider a particle $j \in J$ at the perimeter of one centered sub-domain (exemplarily denoted by a red dot in Fig. 4) as well as the distribution of the respective sub-domains to three GPUs (as denoted at the bottom of Fig. 4). The neighbors of j within $2h$ are not only located in cells of its own sub-domain (covered by GPU 1 in Fig. 4), but also in cells of the neighboring sub-domain (covered by GPU 0). Since the sub-domains are distributed to different GPUs, the neighbors of j are located in a distinct device and, hence, a different memory pool. This hinders fast neighbor access.*

In order to accelerate neighbor access, each GPU should therefore hold a copy of the data located at the *edge* of its adjacent sub-domains, i.e. all cells within $2h$ at the perimeter of a sub-domain (highlighted in colors in Fig. 4). These edges are also referred to as *halo* of a sub-domain. An example of a domain distribution with halo exchange is illustrated in Fig. 5: The domain is split into three sub-domains and the colored edges represent the halos of the sub-domains. Each sub-domain holds a copy of the edge of its adjacent sub-domain(s), which allows for fast neighbor access.

Finally, the inherent moving nature of the particles means that particles do not stay within "their" respective grid, but freely move in space – requiring to frequently update the corresponding workload of the respective GPUs. Overall, these characteristics make load balancing a complex task and determining a sophisticated load balancing strategy is the key to achieve optimal performance.

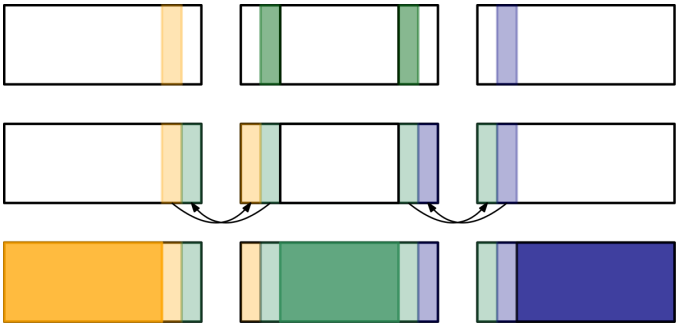


Fig. 5: Halo exchange between three sub-domains.

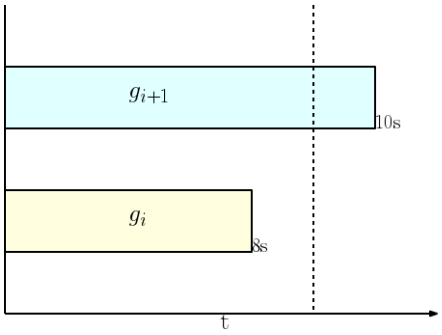


Fig. 6: Unbalanced state in a system composed of two GPUs.

Unfortunately, rather few works exist on this subject thus far. One solution has been presented in the work of [14]. Here, a simple a posteriori load balancing system is introduced, which shows high robustness in the performed test cases. However, the proposed load balancing introduces an overhead, mainly caused by moving edges of one GPU to another. In the work of [13] a similar method is presented – yielding the same disadvantages. Besides that, load balancing for particle-based simulations on multi-GPU architectures has also been considered for classical *Molecular Dynamics* (MD, see e.g. [15], [16]). But MD differs from SPH simulations and usually deals with smaller motions and a wider variety of particles – making the load balancing tasks significantly different. Hence, how to employ an efficient distribution of the workload of SPH simulations to the respective GPUs remains an open question.

IV. Advanced Load Balancing

In this work, we propose an approach for an advanced distribution of the SPH simulation workload to GPU architectures. To this end, we first outline the general idea of our solution. Afterwards, important implementation details are covered, which need to be considered in order to minimize the overhead.

A. General Idea

In any SPH implementation, many unpredictable factors, such as the fluid movement, may influence the computation time during run-time. Hence, an advanced a posteriori load balancing methodology needs to be chosen. In our solution, first an initial decomposition of the domain is calculated. More precisely, the initial number of particles per GPU is approximated by

$$N_i = \frac{N_t}{N_g}, \tag{5}$$

where N_i is the initial amount of particles per GPU, N_t the total amount of particles, and N_g the number of GPUs.

This distribution is used to compute the first time steps, while recording the amount of time spent by each GPU. After an update interval of n time steps, the algorithm iterates through the time data collected from each GPU. Then, the *relative runtime difference* d_r between two neighboring devices g_i and g_{i+1} is determined by

$$d_r = \frac{r_{i+1} - r_i}{r_i}, \tag{6}$$

where d_r is the relative runtime difference, r_i the runtime of g_i, and r_{i+1} the runtime of g_{i+1}.

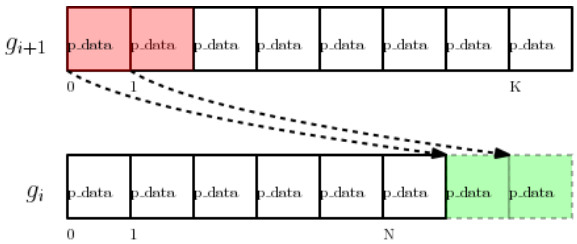
If d_r is larger than a threshold factor p, a *domain decomposition correction* from g_i to $g_i + 1$ is applied, i.e. one *edge* (as introduced in Section III) is moved from the GPU with the longer execution time to the GPU with the shorter one. If d_r is smaller than $-p$, a *domain decomposition correction* from $g_i + 1$ to g_i is applied. This process of corrections is repeated every n further time steps again. Algorithm 1 explains this procedure in pseudocode.

Algorithm 1 Balance

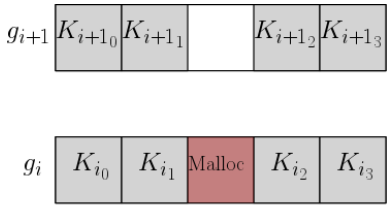
```
n ← update interval
p ← threshold
K ← time steps
N_g ← number of GPUs
for k ∈ {1, ..., K} do
  Calculate Time Step
  if k mod n = 0 then
    for each GPU g_i ∈ {g_0, ..., g_{N_g}} do
      if RuntimeDifference(g_i, g_{i+1}) > p then
        Decomposition Correction(g_i, g_{i+1})
      end if
      if RuntimeDifference(g_i, g_{i+1}) < −p then
        Decomposition Correction(g_{i+1}, g_i)
      end if
    end for
  end if
  Sort Particles
end for
```

Example 4. *Consider the simplest case for a* domain decomposition correction*: a multi-GPU architecture composed of two GPUs as illustrated in Fig. 6. GPU g_i required only 8 seconds to compute n time steps, while GPU g_{i+1} required 10 seconds. Hence, a relative time difference of $d_r = 0.25$ results – yielding an unbalanced state. In order to balance that state, one* edge *of the virtual search grid is shifted from g_{i+1} to g_i.*

The basic concept of this load balancing strategy is rather straight-forward. However, taking a more detailed look into GPU and CUDA architectures, applying such a domain decomposition correction with a minimum amount of overhead is a non-trivial task. In fact, in our implementation each GPU internally stores its particle data consecutively in an array. Within that array, the particle data are kept sorted, in order



(a) Re-allocation of memory during a decomposition correction.



(b) Resulting synchronization step.

Fig. 7: Overhead caused by re-allocations.

maintain cell positions of the search grid. Hence, particles at the perimeter of the domain are also kept at the perimeter of the array. This requires re-allocations of the respective array memories when a domain decomposition correction (and, hence, a re-balancing) and an *exchange of halos* is conducted – accomplished by a *cudaMalloc*-call in the considered CUDA architecture. Unfortunately, this is a synchronous call which causes all GPUs to synchronize. Overall, this results in a major loss in performance.

Example 5. *Consider again a multi-GPU architecture composed of two GPUs and assume that data is currently allocated in arrays as illustrated in Fig. 7a. Now, the particles of the left edge of the sub-domain covered by g_{i+1} (stored in the front of the array of g_{i+1}) should be shifted to the right edge of the sub-domain covered by g_i. This requires the allocation of new memory to the array storing the particles of g_i.*

This, however, significantly affects the execution of the respective computations as illustrated in Fig. 7b. Here, kernel functions (denoted by K_{i_0}, K_{i_1}, ...) are executed in parallel among both GPUs, until cudaMalloc() causes synchronization. Then, all computations have to pause in order to synchronize. Assuming larger multi-GPU systems composed of hundreds of GPUs which frequently need to allocate new memory, this easily accumulates to a severe number of synchronization steps in which all GPUs (even if they are not affected) have to pause – a significant loss in performance.

Obviously, such a loss in performance should be avoided. For that purpose, two different approaches are proposed which are introduced and discussed in the next sections.

B. Using Internal Cache

A first solution employs an additional local cache to shift data (as illustrated in Fig. 8). Here, *cudaMemcpyAsync* from the CUDA API is used, which allows for an efficient asynchronous copy of data from the local data array to a local cache. After copying, the pointers between the array and the cache are swapped – leaving the local array with the reduced data and the cache with the original objects. During that process, the capacity of both, the data array and cache are only reduced when the capacity is significantly larger than the used memory. Instead, only the internal number of elements is modified without actually freeing any memory. The benefit of that method is the minimized synchronization caused by allocating new memory, since *cudaMalloc* only needs to be called when the amount of data to be processed in one GPU is larger than the initial amount N_i of particles covered per GPU. This drastic reduction of synchronization efforts results in a noticeably improved performance. In case new data needs to be shifted back to the device, the local data array is examined if the new data fits to the already allocated data chunk. If that check fails, new memory needs to be allocated.

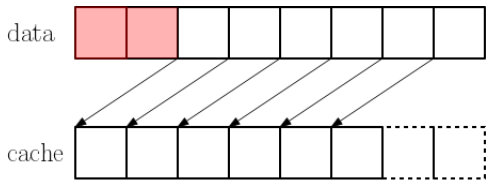


Fig. 8: Avoiding synchronization using internal cache.

Example 6. *Consider again the scenario discussed in Example 5. Employing the cache implementation as described above, the capacity of g_{i+1} array is not reduced. Instead, data is just shifted by one block. Assuming that one edge of g_i needs to be shifted back in the next domain decomposition correction, e.g. due to unpredictable fluid structure behavior, memory allocation can be avoided since data has not been freed during the previous domain decomposition correction.*

The obvious drawback of this method is the demand for copying pointers and the increase of memory consumption. However, as previously discussed, particle data always needs to be kept sorted. For that purpose, a radix sort is employed, which is one of the fastest sorting algorithms on GPUs. Since radix sort is typically an out-of-place sorting algorithm, a temporary storage buffer is required. Therefore, the memory consumption of radix sort is $O(2N + P)$, where N is the length of the input data to be sorted and P the number of streaming multiprocessors on the device. For that purpose, the cache can be employed as a temporary storage, which leaves the high watermark memory consumption unchanged by radix sort. However, in case memory requirement is the bottleneck for a given computation, a second solution with reduced memory consumption is proposed next.

C. Using Pointers

The main drawback of the cache method is the increased memory consumption. To avoid that memory overhead, an alternative method is proposed in which data is not shifted using an additional cache, but by shifting pointers to the beginning of the data block (as illustrated in Fig. 9). For example, in case the first two elements should be deleted, the pointer is incremented accordingly. Of course, data is not only removed from the front but also from the back, when data needs to be transferred from GPU_i to GPU_{i+1}. In that case, the pointer to end is moved accordingly.

Similar to the cache implementation, no memory needs to be freed – leaving the capacity of the array unchanged and

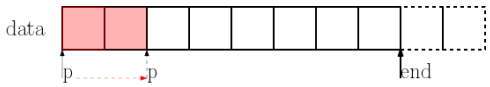


Fig. 9: Avoiding synchronization using pointers.

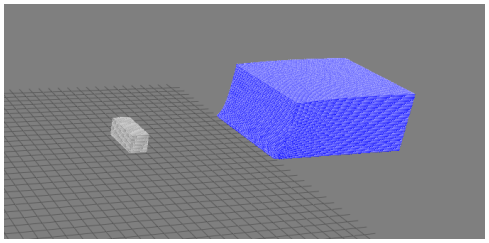


Fig. 10: Sketch of the considered scenario (dam break).

avoiding frequent memory allocations. However, in contrast to the cache implementation, free memory blocks can occur in the front or in the back of the array when the pointer-solution is applied. Because of this, there are frequent checks for unoccupied space in the front and in the back of the array which can be used to accordingly add data. Only when there is not enough free space available, i.e. the amount of needed memory is larger than the original setup N_i, new memory is allocated.

Clearly, the advantage of this method is that internal cache can be omitted, which results in a decreased memory consumption. However, the inherent out-of-place behavior of radix sort occupies a temporary storage buffer for sorting. Therefore, the high watermark memory consumption during sorting is defined as $O(2N + P)$. Efficient in-place sorting methodologies on GPUs do not provide comparable time complexity. For example, comparing the so-called *bitonic sort* for GPUs as initially proposed by Peters et al. [17]. Bitonic sort is an efficient comparison-based in-place sorting algorithm for CUDA. It offers a time complexity of $O(N \log^2 N)$ compared to a complexity of $O(N \log N)$ for radix sort. Hence, the decreased memory consumption is traded off by a weaker sorting performance.

V. Experimental Evaluations

In order to evaluate the performance of the proposed load balancing approaches for the SPH technique, we have implemented the methods described above in C++ for an industrial SPH solver and conducted experiments whose results are summarized in this section. As a test case, we considered a dam break scenario with a rotating obstacle in the middle as sketched in Fig. 10. To evaluate the scalability of the method, this scenario has been considered using 385k, 2.38mio, and 5.26mio particles. All evaluations have been conducted on GPU systems composed of Nvidia GTX 1080 Ti, which contain 3584 CUDA cores with a memory bandwidth of 484 GB/s. The source code was compiled on Ubuntu v16.04 using gcc v4.5.3 and the CUDA Toolkit v8.0.

We are comparing both, the cache and the pointer implementation (as introduced in Section IV-B and Section IV-C, respectively) with respect to speedup and efficiency.

The speedup is defined as

$$S = \frac{T_s(n)}{T_p(n,p)} \tag{7}$$

TABLE I: Results obtained by the experimental evaluation.

(a) Using the cache implementation from Section IV-B

	Speedup			Efficiency		
# GPUs	130k	2,40m	5.26m	130k	2,40m	5.26m
1 GPU	1	1	1	1	1	1
2 GPU	1.56	1.64	1.68	0.78	0.82	0.84
3 GPU	2.05	2.25	2.31	0.68	0.75	0.77

(b) Using the pointer implementation from Section IV-C

	Speedup			Efficiency		
# GPUs	130k	2,40m	5.26m	130k	2,40m	5.26m
1 GPU	1	1	1	1	1	1
2 GPU	1.61	1.70	1.79	0.81	0.85	0.90
3 GPU	2.14	2.36	2.45	0.71	0.79	0.82

where S is the speedup, n the size of the input (amount of particles in the system), T_S the execution time of the single GPU implementation, T_p the execution time of the multi-GPU implementation, and p the number of used GPUs. The efficiency is defined as

$$E = \frac{S}{p} \tag{8}$$

where E is the efficiency, S the speedup, and p the number of GPUs.

Table Ia shows the speedup and efficiency using the cache implementation. The values show that with larger number of particles, the achieved speedup increases. This is inherently the case, since the more particles that need to be computed the smaller the proportion spent on communication between GPUs.

Table Ib shows the speedup and efficiency using the pointers implementation. The values show that in the present test example, the pointer implementation is superior in all setups compared to the cache implementation. This is mainly because the dam break is an asymmetrical problem, where particles are clustered at one side in the beginning of the process and equally distributed in the end. Therefore, frequent communication between GPUs is necessary, since particles move quickly between the respective sub-domains. This is suitable for the pointer implementation which needs less overhead by copying pointers to local cache and, hence, results in an improved speedup.

The presented results also show that the methods proposed in this work, scale well for smaller amount of particles. For example, an efficiency of 0.79 is achieved when using 2,4mio particles on 3 GPUs, but also for a tiny number of particles of 130k on 3 GPUs, an efficiency of 0.71 is achieved.

VI. Conclusion and Future Work

In this work, we presented a load balanced multi-GPU CUDA-based implementation of an industrial SPH solver. To this end, we employed a sophisticated domain decomposition strategy and proposed a load balancing methodology, which also scales well for smaller amount of particles. Important implementation details based on the CUDA architecture are provided, which reduce the synchronization overhead and, therefore, result in an increased efficiency. Future work includes the implementation of a multi-node solution, hence, the introduction of an additional level of parallelization, and further optimization on the domain decomposition strategy to achieve further speedups.

ACKNOWLEDGMENT

This work has been supported by the Austrian Research Promotion Agency (FFG) within the project Industrienahe Dissertationen 2016 under grant no. 860194.

REFERENCES

[1] J. Monaghan, "Smoothed particle hydrodynamics and its diverse applications," *Annual Review of Fluid Mechanics*, vol. 44, pp. 323–346, 2012.

[2] M. Dhaubhadel, "Review: Cfd applications in the automotive industry," *Int. J. Fluids Eng*, vol. 118(4), pp. 647–653, 1996.

[3] H. Yoon, S. Koshizuka, and Y. Oka, "Direct calculation of bubble growth, departure and rise in nucleate pool boiling," *Int. J. Multiphase Flow*, vol. 27, pp. 277–298, 2001.

[4] D. Hietel, K. Steiner, and J. Struckmeier, "A finite volume particle method for compressible flows," *Mathematical Models and Methods in Applied Science*, vol. 10, pp. 1363–1382, 2000.

[5] R. Gingoldand and J. Monaghan, "Smoothed particle hydrodynamics - theory and application to non-spherical star," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, 1977.

[6] Z. Zhang and Q. Chen, "Comparison of the eulerian and lagrangian methods for pedicting particle transport in enclosed spaces," *Atmospheric Environment*, vol. 41, pp. 5236–5248, 2007.

[7] M. Shadloo, G. Oger, and D. Touze, "Smoothed particle hydrodynamocs method for fluid flows, towards industrial applications: Motivations, current state, and challenges," *Int. J. Computers and Fluids*, vol. 136, pp. 11–34, 2016.

[8] J. Monaghan, "Smoothed particle hydrodynamics," *Rep. Prog. Phys*, vol. 68, pp. 1703–1759, 2005.

[9] M. Liu and G. Liu, "Smoothed particle hydrodynamics (sph): an overview and recent developments," *Arch. Comput. Methods Eng*, vol. 17, pp. 25–76, 2010.

[10] A. Kolb and N. Cuntz, "Dynamic particle coupling for gpu-based fluid simulation," in *Int. Proc. of the 18th Symposium on Simulation Technique*, 2005, pp. 722–727.

[11] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on gpus," in *Proc. 5th Int. Conf. Computer Graphics*, 2007, pp. 63–70.

[12] A. Herault, G. Bilotta, and R. Dalrymple, "Sph on gpu with cuda," *Int. J. Hydraulic Research*, vol. 48, pp. 74–79, 2010.

[13] J. Dominguez, A. Crespo, and B. Rogers, "New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters," *Int. J. Computer Physics Communications*, vol. 184, pp. 1848–1860, 2013.

[14] E. Rustico, G. Bilotta, A. Herault, C. Negro, and G. Gallo, "Advances in multi-gpu smoothed particle hydrodynamics simulations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, 2014.

[15] C. Trott, L. Winterfeld, and P. Crozier, "General-purpose molecular dynamics simulations on gpu-based clusters," *Int. J. Computer Physics Communications*, 2010.

[16] Q. Wu, C. Yang, T. Tang, and K. Lu, "Fast parallel cutoff pair interactions for molecular dynamics on heterogeneous systems," *Tsinghua Science and Technology*, vol. 17, pp. 265–277, 2012.

[17] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," *Concurrency and Computation Practice and Experience*, vol. 23, pp. 681–693, 2011.