The Visual Computer (2023) 39:983–995
https://doi.org/10.1007/s00371-021-02379-w

ORIGINAL ARTICLE

Optimizations for predictive–corrective particle-based fluid simulation on GPU

Samuel Carensac[1] · Nicolas Pronost[1] · Saïda Bouakaz[1]

Accepted: 5 December 2021 / Published online: 10 January 2022

Abstract
The use of particles-based simulations to produce fluid animations is nowadays a frequently used method by both the industrial and research sectors. Although there are many variations of the smoothed particle hydrodynamics (SPH) algorithm currently being used, they all have the common characteristic of being highly parallel in nature. They are therefore frequently implemented on graphics processing units (GPUs) to benefit of high computation capacities of modern GPUs. However, such optimizations require specific optimizations to make use of the full capacity of the GPU, with sometimes optimizations being contradictory to optimizations used in CPU implementations. In this paper, we explored various optimizations on a GPU implementation of a recent particle-based fluid simulation algorithm using an iterative pressure solver. In particular, we focused on CPU optimizations that have not been thoroughly studied for GPU implementations: the indexing for the neighbor's structure, the frequency of the sorting of the fluid particles, the use of lookup tables for the kernel function computations and the use of a warm-start to improve the performance of the iterative pressure solver. We show that some of these optimizations are only effective for very specific hardware configurations and sometimes even impact the performance negatively. We also show that the warm-start reduces the computation time but introduces a cyclic instability in the simulation. We propose a solution to reduce this instability without requiring to modify the implementation of the fluid algorithm.

Keywords Fluid simulation · SPH · Animation · GPU

1 Introduction

Nowadays, the use of simulations to produce fluid animations is widespread due to the high level of visual realism they allow. One of the main challenges limiting the use of simulation-based fluid animations is the high computation time needed in particular for lengthy animations or animations covering a large space. Among the multiple models to simulate a fluid, the Lagrangian particle-based approach has received a high level of interest for its ability to handle complex virtual environments containing various objects and fluid with extremely variable properties. The most common Lagrangian algorithm used in fluid simulation, the Smoothed Particles Hydrodynamics (SPH), requires the use of very small simulation steps which greatly limits the possibility of interactive simulations. This limitation is due to the direct computation of the pressure forces between the fluid particles that is unstable for large simulation steps. To reduce the overall computation time, an approach consists in replacing the direct computation by iterative processes that progressively compute a better approximation of the pressure forces, thus improving the fluid stability. This approach allows for the use of larger simulation steps by compensating the increased computation time necessary for each simulation step by lowering the frequency at which the computations are done. This approach also allows for a better control of the fluid properties, such as the fluid incompressibility, through the use of thresholds that the user can select to obtain the desired level of stability required in their experiments. This approach is used in the predictive–corrective SPH algorithm (PCISPH) [27] through a predictive–corrective process that iterates within one simulation step until the density of the fluid is within a given margin of error from the desired density. This algo-

✉ Nicolas Pronost
nicolas.pronost@univ-lyon1.fr

Samuel Carensac
samuel.carensac.research@gmail.com

Saïda Bouakaz
saida.bouakaz@univ-lyon1.fr

1 Université de Lyon, Université Claude Bernard Lyon 1, CNRS LIRIS UMR 5205, Villeurbanne, France

rithm has been further extended in the divergence-free SPH (DFSPH) [6] by adding a second predictive–corrective process which ensures that the average divergence within the fluid stays null. However, even those more advanced algorithms are still far from producing interactive large-scale simulations.

The SPH algorithm requires the computation of physical properties such as the density at every fluid particle for each simulation step. Consequently, the SPH algorithm, regardless of the pressure solver, is, usually, naturally parallelizable. This is why, most research publications using the SPH algorithm present results using multi-threaded CPU implementations. However, even recent high-end CPU usually only have 52 or less cores, which is way lower than the hundreds of thousands of particles used in a single simulation. Looking at current GPU, we can see that even relatively cheap consumer-grade cards offer thousands of cores making them highly desirable for highly parallelizable workloads. This is why, in this last decade, numerous works proposed GPU and multi-GPU implementations of SPH algorithms that are 20 times faster or more than a sequential CPU implementation [13–15]. Although those GPU implementations can be extremely similar to the CPU implementations, the technical differences between CPU and GPU can lead to some optimizations that are efficient on CPU but have a wildly different impact on a GPU implementation. This paper aims to study the impact of various optimizations on a GPU implementation of a SPH algorithm that is using an iterative pressure solver, specifically the DFSPH algorithm.

2 Previous works

The Lagrangian approach to physically simulate fluids is nowadays commonly used through the SPH algorithm for its simplicity of handling free surfaces. The initial SPH algorithm [24] has received multiple improvements, notably the commonly used $delta$-SPH algorithm [2] which adds a diffusive term to improve the stability of the pressure field in the fluid. The ISPH algorithm [10] departs further from the initial SPH algorithm by using a linear system to compute the pressure in the fluid improving further the stability by allowing true incompressibility of the fluid at the cost of the scalability of the algorithm. Another approach proposed in the PCISPH algorithm [27] is to replace the direct computation of the pressure forces by an iterative predictive–corrective process to enforce a constant density condition on the fluid. This approach was extended in the IISPH [20] and DFSPH [6], the later adding a second predictive–corrective process enforcing a divergence-free condition further improving the stability of the fluid. The high stability of the DFSPH enables the use of larger simulations steps lowering the computation time needed for the whole simulation. The higher stability of this algorithm has been used to handle simulations such as highly viscous fluids [7,29] or recent boundary models [8,30].

To our knowledge, there are only few works studying the optimizations of SPH implementations that are independent from a specific SPH algorithm [9,11]. A large number of those optimizations have been focused on the algorithms and data structures used for the neighbors search. The two most common structures used to find the particles neighborhoods are a hash-table or a cell linked list using a uniform grid [19] but some works have explored the use of multi-resolution structures [31]. One large difference between existing SPH implementations is whether each particle stores its neighbors or not. This choice is mostly a trade-off between computation time and memory space, especially for predictive–corrective SPH algorithms [9,19]. Some recent works achieved to reduce the computation time used for the neighbor list construction using a Verlet list [9,32] or to compress the neighbor list to reduce the memory impact while not having to re-explore the data structure [3]. We refer the reader to the state of the arts [21,22] for a proper overview of the structures employed in the neighbor search. An optimization often associated with the neighbor search is the use of a space filling curve to sort the particles. Recent works prefer to use a Morton curve (also called Z curve) to sort the particles instead of the linear XYZ curve to improve the cache hit rate [19]. Sometimes the Hilbert curve is also used but does not seem to be superior to the Morton curve [3]. The use of each curve can be further optimized by accessing consecutive cells as a single block when looking for a particle neighborhood. This can be done trivially for the XYZ curve [11] or by using specialized algorithms like the BigMin-LitMax algorithm for the Morton curve [3]. However, we were unable to find existing works replicating these experiments on a GPU implementation.

The execution times can be reduced by using appropriate data structures and memory management. One of such optimizations used in nearly every implementation is the use of a Structure of Array (SoA) to store the particle data. This simple optimization can improve the performance of particle-based systems on both CPU [4] and GPU [9]. It is possible to further improve the data structure by using wider primitive types that fit the size of a unitary data access for a given architecture. This can be done by simply padding the data structure or even better by using the larger primitive type to store data that are often accessed together [9,11]. In the DFSPH approach, Bender et al. [6] propose to use a lookup table instead of directly computing the values of the kernel function and report a 30% reduction in the computation times for a multi-threaded CPU implementation. They also use a warm-start mechanism for the predictive–corrective loop. This optimization is commonly used for iterative processes and is also used by other PCISPH-based algorithms [20].

However, these works do not study the exact performance gains offered by that system.

Some optimizations were proposed to fit the existing memory types and access patterns of current GPUs. Goswami et al. [14] have proposed to use the higher performance shared memory by transferring the data from the global memory into it before running the desired computation. This technique has been improved by Huang et al. [18] by properly distributing the particles between thread groups. When using an approach storing the neighbors list for each particle, we can improve the performance by interleaving the neighbors of each particle [9]. This technique improves the performance as modern GPUs stream block locally adjacent data over warps of 32 threads. Billota et al. show that it is even possible to improve the performance even further by interleaving groups of neighbors [9]. The performance can be further improved by using a multi-GPU system. Each GPU is given a *slice*, preferably dynamic [25,28], of the fluid to handle and data is transferred with the CPU depending on the particle displacement during a simulation step. Specialized data structures are used due to the need of synchronization and transfer between each GPU [9,28].

3 Divergence-free smoothed particle hydrodynamics (DFSPH) on GPU

3.1 DFSPH overview

In this work, we use the DFSPH algorithm proposed by Bender et al. [6]. We must note that we did not use the Courant–Friedrich–Levy (CFL) rule, used to determine the maximal time-step, because we did not want dynamic time-steps. Indeed, as we need a high level of repeatability between experiments in order to test the optimizations, a constant time-step is necessary. Algorithm 1 shows the details of one simulation step and Algorithms 2 and 3 describe the predictive–corrective process, respectively, for the constant density solver and the divergence-free solver. The algorithms for the two predictive–corrective solvers show how the warm-start mechanism is integrated (lines 2 to 4 and line 11 of both algorithms).

We have chosen the DFSPH algorithm for its stability and speed of simulation while having relatively large simulation steps. The DFSPH benefits from a large stability increase relatively to the previous IISPH algorithm. We refer the reader to the original publication for more details on the DFSPH algorithm [6].

Algorithm 1 DFSPH step

```
1:  for all particles i do
2:      find Neighborhood $N_i$
3:  for all particles i do
4:      compute densities $\rho_i = \sum m_j W_{ij}$
5:      compute factors $\alpha_i = \frac{\rho_i}{\|\sum m_j \nabla W_{ij}\|^2 + \sum \|m_j \nabla W_{ij}\|^2}$
6:  correctDivergenceError($\alpha$, v)
7:  for all particles i do
8:      compute non-pressure forces acceleration $acc_i$
9:  for all particles i do
10:     $v_i \mathrel{+}= \Delta t * acc_i$
11: correctDensityError($\alpha$, v)
12: for all particles i do
13:     $x_i \mathrel{+}= \Delta t * v_i$
```

Algorithm 2 Constant density solver

```
1:  function CORRECTDENSITYERROR($\alpha$, v)
2:      for all particles i do
3:          apply warm-start $K_i = \kappa_i, K_j = \kappa_j$
4:          correct velocity $v_i \mathrel{-}= \Delta t \sum m_j (\frac{K_i}{\rho_j} + \frac{K_j}{\rho_j}) \nabla W_{ij}$
5:      for all particles i do
6:          predict $\rho_i^* = \rho_i + \Delta t \sum m_j (v_i^* - v_j^*) \nabla W_{ij}$
7:      while $(\rho_{avg}^* - \rho_0 > \eta) \vee (iter < 2)$ do
8:          for all particles i do
9:              $K_i = \frac{\rho_i^* - \rho_0}{\Delta t^2} \alpha_i, K_j = \frac{\rho_j^* - \rho_0}{\Delta t^2} \alpha_j$
10:             update $v_i \mathrel{-}= \Delta t \sum m_j (\frac{K_i}{\rho_j} + \frac{K_j}{\rho_j}) \nabla W_{ij}$
11:             update warm-start $\kappa_i^v \mathrel{+}= K_i^v$
12:         for all particles i do
13:             predict $\rho_i^* = \rho_i + \Delta t \sum m_j (v_i^* - v_j^*) \nabla W_{ij}$
14:         compute $\rho_{avg}^* = \frac{1}{N} \sum \rho_i^*$
```

Algorithm 3 Divergence-free solver

```
1:  function CORRECTDIVERGENCEERROR($\alpha$, v)
2:      for all particles i do
3:          apply warm-start $K_i^v = \kappa_i^v, K_j^v = \kappa_j^v$
4:          correct velocity $v_i \mathrel{-}= \Delta t \sum m_j (\frac{K_i^v}{\rho_j} + \frac{K_j^v}{\rho_j}) \nabla W_{ij}$
5:      for all particles i do
6:          predict $\frac{D\rho_i}{Dt} = \sum m_j (v_i - v_j) \nabla W_{ij}$
7:      while $((\frac{D\rho}{Dt})_{avg} > \eta^v) \vee (iter < 1)$ do
8:          for all particles i do
9:              $K_i^v = \frac{1}{\Delta t} \frac{D\rho_i}{Dt} \alpha_i, K_j^v = \frac{1}{\Delta t} \frac{D\rho_i}{Dt} \alpha_j$
10:             predict $v_i \mathrel{-}= \Delta t \sum m_j (\frac{K_i^v}{\rho_j} + \frac{K_j^v}{\rho_j}) \nabla W_{ij}$
11:             update warm-start $\kappa_i \mathrel{+}= K_i$
12:         for all particles i do
13:             predict $\frac{D\rho_i}{Dt} = \sum m_j (v_i - v_j) \nabla W_{ij}$
14:         compute $(\frac{D\rho}{Dt})_{avg} = \frac{1}{N} \sum \frac{D\rho_i}{Dt}$
```

3.2 Implementation

For the experiments of this paper, we have implemented the DFSPH algorithm on GPU with CUDA. Our implementation is based on the CPU OpenMP implementation of the DFSPH algorithm found in the SPlisHSPlasH library [5]. To be specific, we used the DFSPH implementation from the 1.3.1 version of the library. We chose this library because the OpenMP structure of the code makes it relatively easy to convert into a GPU implementation. Moreover, the library contains a physics engine, multiple algorithms for surface tension, collisions and viscosity and also multiple SPH algorithms making comparisons and tests easier.

However, our implementation differs from a CPU implementation on the following points. The SPlisHSPlasH library uses an external library to handle the neighbors search (Algorithm 1, line 2). This library uses a compact hashing structure. We implemented a cell linked list approach using a counting sort [17] to fill the structure. Also, as mentioned earlier, our implementation does not use the dynamic time-step using the CFL rule. We should also note that we reorganized various parts of the computation to minimize the number of GPU kernels and the number of global memory accesses.

On top of these modifications, we also implemented two optimizations that may have an impact on the simulations. The first one is the interleaving of the neighbors inside the structure storing the neighbors of all the particles. This simple optimization can offer a large improvement in the performance, although we did not used the group interleaving mentioned in [9]. The second one is the use of the CUDA-OpenGL interoperability which allows the usage in the CUDA kernels of buffers allocated by OpenGL. We allocated the position and velocity buffers this way since they are used to render the fluid.

The last well-studied optimization we implemented is the padding of the vector structure used to store the position and the velocity [9,11]. Since the GPU accesses the data in packets of 4, 8 or 16 bytes, it is better to use structures that have one of these sizes. But a position or a velocity is only composed of 3 floating points in a 3D simulation, meaning that when using standard single floating point precision (i.e., 4 bytes), the total size of the structure is 12 bytes thus wasting 4 bytes with each access. The optimization consists in adding a fourth value to the structure where an often used value will be stored. This value gets then loaded for free. In practice, the mass of a particle can be added to the position structure and its density to the velocity structure. We have implemented this optimization in our framework and tested with both a single floating point precision and a double floating point precision and with three GPUs: a consumer-grade GeForce GTX 1070, a more recent GeForce GTX 2070 Super and a professional-grade Quadro 4000k. The scenario used for this study is a dam-break with approximately 288k particles over 5 seconds of simulation. We report the average computation times in Table 1 and a frame of each configuration is shown in Fig. 1. We can see that the use of the padded structure does not bring benefits in most cases. When using a single floating point precision, this optimization reduces the performances by 10% for the 2070 and the Quadro cards and has no impact on the 1070. In double precision, there is no impact for the Quadro and 1070, but on the 2070 card, the optimization improves the computation time by 15%. Since the single floating point precision provides significantly better performance, especially on consumer-grade cards, most of the following experiments will use this precision. And therefore, the padding optimization will not be used. Our results conflict significantly with the results of Bilotta et al. [9] that showed a 50% reduction of the computation time on a GTX 750 card with single precision. But as we can see in our experiments, the outcome greatly depends on the card and recent cards show less benefits for this optimization. In regards to these conclusions, we chose not to use this optimization in the remainder of our simulations.

In our experiments, we took the original multi-thread CPU implementation from the SPlisHSPlasH library [5] as reference. These reference simulations were obtained using a double Intel(R) Xeon(R) E5-2623v3 (total of 8 cores) and using a more recent Intel(R) Xeon(R) W-2255 (10 cores). The results for the dam-break experiment are reported in Table 1. We can see that there is a very significant improvement by using a GPU implementation. Even the Quadro 4000k, released in 2013, equals or outperforms a recent high-grade CPU. If we compare the most recent of each category, the GTX 2070 Super and Xeon W-2255, both of them released in 2019, we can see an improvement factor of 4 even when using double precision. However, NVidia's consumer-grade GPU,

Table 1 Average computation time (in seconds) of a dam-break simulation with 288k particles on CPU and GPU, depending on the floating point precision and the usage of the padding optimization

	32 bits	32 bits padded	64 bits	64 bits padded
GPU GeForce 1070	84	82	216	216
GPU GeForce 2070 Super	31.6	34.9	219	196
GPU Quadro 4000k	456	525	752	791
CPU Xeon E5-2623v3	–	–	1198	–
CPU Xeon W-2255	–	–	810	–

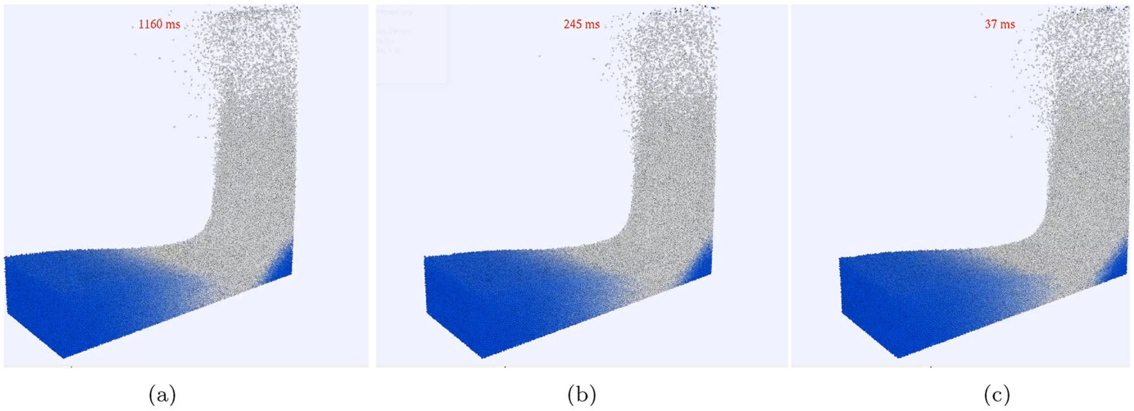


Fig. 1 Frames of a dam-break simulation with $288k$ particles on **a** CPU (Xeon W-2255) 64 bits, **b** GPU (GTX 2070 Super) 64 bits and **c** GPU (GTX 2070 Super) 32 bits

the GeForce lineup, is made to use single precision floating points. With that precision, we observe an improvement factor of 25 compared to the multi-threaded CPU implementation using hardware of the same generation.

4 Optimizations

In this section, we will study various optimizations that have yet to be studied on GPU and evaluate their impact on our implementation. Our experiments use a dam-break scenario with the following dimensions : a (16;8;4) solid box containing the fluid which is initialized as a (6;6;4) cube with particles organized on a regular grid. All dimensions are given in meters and the vertical axis is the Y-axis. The fluid particle size is fixed at $0.025m$. This setup gives a simulation with around 1.1 million fluid particles and 420k solid particles (using Akinci et al. boundaries model [1]). Unless specified otherwise the following parameters are applied : a fixed time-step of $3ms$, a density of $1000kg/m^3$, a viscosity of 0.01, a density threshold of 0.01% and a divergence threshold of 0.1%.

Every value reported in our experiments is an average over 5 seconds of simulation. Also, as the number of iterations of the density and divergence solvers varies between simulations, it is required to normalize them before reporting the average. On average this simulation scenario uses 12 density iterations and 2 divergence iterations every simulation step and the values reported are normalized to fit those values. Finally, with these particular parameters, usually around 120 fluid particles will tunnel through the border of the solid rectangle. These particles are removed from the simulation, although the time taken to remove them is not taken into account in the timings. This loss of particles is due to our choice of enforcing a constant time-step and to not fine-tune the parameters used for the boundary handling. These choices have been made to make the experiment more repeatable and easy to reproduce. Similarly, our results only report the computation times used by the physical simulation, the rendering times are not reported.

We used two different graphic cards : a consumer-grade GeForce GTX 1070 and a professional-grade Quadro 4000k. Unless specified otherwise the presented results are obtained with a single floating point precision implementation using the GeForce card.

4.1 Neighbor structure index

Our first interest is to study the impact on computation time of the type of space filling curve used for the neighbor search structure. This optimization affects the neighbors search step (see Algorithm 1, line 2). We compare computation times obtained with the two most common curves : the linear XYZ curve [11] and the Morton Z-order curve [19], but we also studied a Hilbert curve [3]. As there are multiple ways to define a Hilbert curve when considering 3 dimensions [16] and we chose to use the one implemented by John Skilling [26] for the ease of reproducibility of our experiments. An illustration of a 2D representation of each of these three curves is given in Fig. 2.

We start by comparing the computation time observed for one simulation step for each curve. Our goal is to evaluate the absolute benefits of the more complex curves without having them penalized by their cost. In practice, we removed the time used to build the neighbor list of each particle and to sort the particle data and the neighbor list following the curve of interest at each simulation step. We also measured the computation time when the order of the particles is randomized at each simulation step.

The results are presented in the first row of Table 2. As expected, we can see that the organized curves perform better. However, we can also see that the difference between each

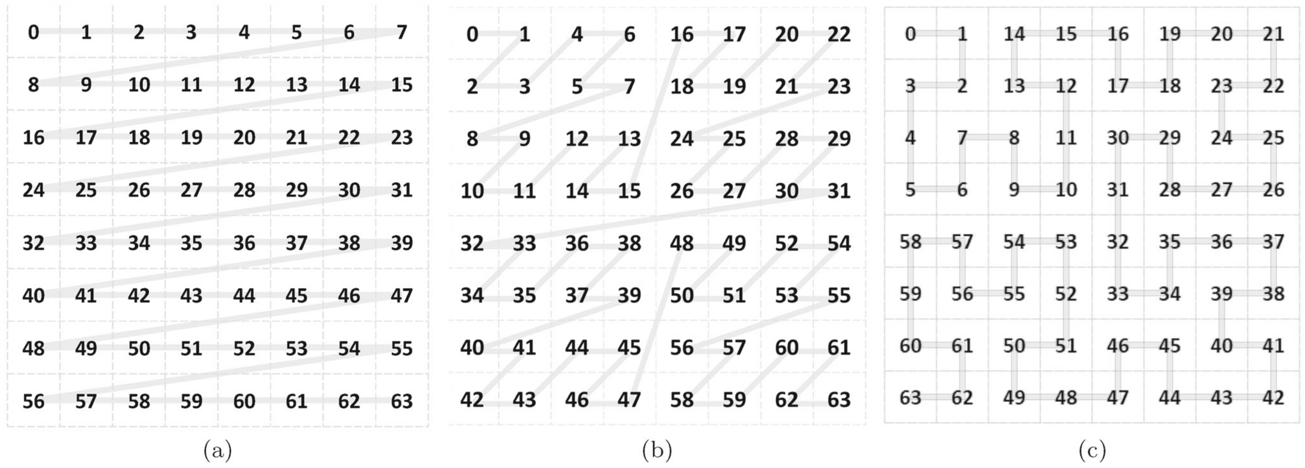


Fig. 2 2D curves corresponding to the indexes used in our experiments: **a** XYZ curve, **b** Morton curve, **c** Hilbert curve

Table 2 Computation time (in ms) for the different space filling curves

	Randomized	XYZ	Morton	Hilbert
GeForce 32 bits neighbors sorted	600.2	294.7	302.8	285.7
GeForce 32 bits neighbors XYZ	XXX	294.7	307.6	298.3
Quadro 32 bits neighbors sorted	XXX	2403.9	2343.7	2170.0
Quadro 32 bits neighbors XYZ	XXX	2403.9	2532.5	2420.1

curve is very different from what one would expect looking at the CPU results reported in previous works [3,19]. On the GeForce card, the Morton curve does not bring any performance gain relatively to a simple XYZ curve. The Hilbert curve fares a bit better as we observed a reduction of approximately 3%. The results are better for two curves when using the Quadro card and in particular, the Hilbert curve offers an improvement of around 10%. Those results are quite lackluster considering that constructing a sorted neighbor list is considerably more complex with those two curves compared to the XYZ curve. The main reason is that the fastest algorithm to explore the 27 cells surrounding a particle is a simple triple nested iteration loop. As such, for the Morton and Hilbert curves more complex algorithms must be used because the cost of sorting the neighbor list after construction overrides any benefit the curve may have brought. Not sorting the neighbor list is not an option as having the neighbor list sorted is really important to obtain better performance. This can be easily seen by looking at the computation times obtained if the order of the neighbors is simply obtained by exploring the neighbors cells by using simple triple nested loops when storing the neighbors (see second row of Table 2).

Table 3 reports the construction times (including the sorting of particles) for the sorted neighbors list of our current implementation when executed on the GeForce 32 bits card. We can observe that the difference in construction time with the complex curves surpasses the benefit we may have observed above for the Morton curve and roughly equals the benefit observed with the Hilbert curve. However, we must consider that we have not optimized the construction of the sorted neighbors list. Currently, we use a simple iteration on each of the 27 neighboring cells, select the one with the lower index not yet explored and we repeat that process until every cell is explored. This is a naive approach and algorithms have been developed to improve that exploration when a Morton curve is used [3]. As our earlier experiments showed no benefit in using the Morton curve, we have not implemented this algorithm. However, as the current total computation time with the Hilbert curve is close to the time obtained with the XYZ curve, it would be interesting to use a better exploration algorithm.

The XYZ curve has one last advantage compared to the others. In our experiment, we added a special XYZ curve denoted as XYZ-advanced (see Table 3). This is a linear curve that uses the fact that the cells are separated in 9 groups of 3 continuous cells to remove one of the iteration loops, as proposed in [11]. This optimization only gives a small reduction of the computation time as long as the 27 cells are explored in the order of the curve when not using that optimization. Maybe, this kind of optimization would have a larger impact with indexes that have chains of cells longer than 3.

In this paper, we refer at the Hilbert curve as "Hilbert-storage" as the indexes of the cells are relatively expensive to compute and as such they are precomputed in a lookup table that uses the XYZ curve to find the corresponding cell values.

Table 3 Neighbors list construction time (in ms) on a Geforce 32 bits card

XYZ	XYZ-advanced	XYZ-storage	Morton	Morton-storage	Hilbert-storage
27.2	25.0	30.6	29.1	34.4	36.4

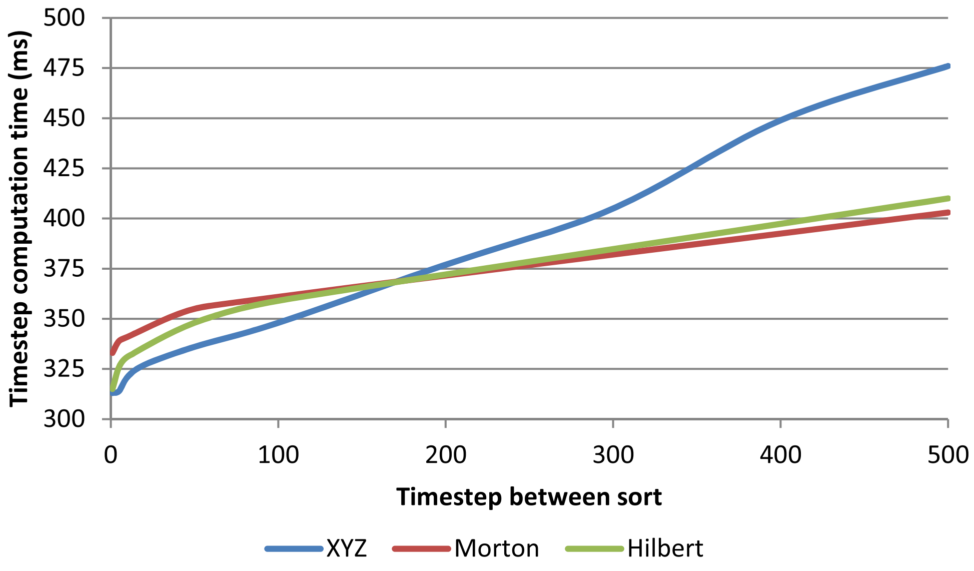


Fig. 3 Evolution of the average computation time depending on the number of time-steps between data sort

The goal was to study if the usage of this lookup table was the reason for the slightly longer construction times when using the Hilbert curve. We implemented the other two curves in the same manner. We can see that the cost of such an operation is around 10% of the construction time for the XYZ curve and 15% for the Morton curve, bringing the Morton curve computation time close to the ones using the Hilbert curve. The difference is due to our simple cell neighbor exploration that will require less memory accesses for the XYZ index than any other index. Still, the cost of precomputing the index is relatively low. This observation is important in particular in the case of curves that are complex to compute such as the studied Hilbert curve. We would like to note that this result implies that such lookup table could be used to make viable any curve no matter the complexity of its construction, and that perhaps another 3D Hilbert curve, or even a novel curve that is impossible to generate procedurally, would give better results.

4.2 Particles sorting frequency

There is one additional parameter we did not consider in the previous experiments that may explain the discrepancy with previous CPU results: the frequency of the sorting of the particle data. This optimization affects again the neighbors search step (see Algorithm 1, line 2). In our experiments, we have observed that sorting the particle data (positions, velocities...) only takes a negligible computation time. For example in our damn-break scenario, sorting all the particle data only takes between $3ms$ and $5ms$, so around $1-2\%$ of the computation step. Existing works rarely specify the sorting frequency of their data. A notable exception is the work of Ihmsen et al. [19] which indicates that sorting the data every 100th steps is sufficient. However, Durand et al. [12] show that sorting at every step still brings some benefit on CPU. The DFSPH implementation provided in the SPlisH-SPlasH library [5] sorts the data every 500th steps. To test that parameter we used the same dam-break scenario and noted the average simulation step computation time for each index for various sort frequencies (see Fig. 3). Interestingly we can observe that the behavior observed with the linear XYZ index is different from the Morton and Hilbert indexes. Although the more complex indexes required more computation time initially due to the complexity of building an ordered neighbor list, we can see that they are more robust to less ordered data. If we use the sorting frequency used in the SPlisH-SPlasH library, we can see that the XYZ index is now $50ms$ slower than the other two. However, we want to note that there is no sorting frequency that gives any significant, if any at all, computation time reduction over just sorting at every simulation step. This is because sorting the data is fast and any reduction in the cache hit rate will significantly worsen the computation time. Consequently, sorting the data at every time-step, or 2 if a gain of around $1.5ms$ is desirable, while using the XYZ index, seems the better configuration.

Table 4 Kernel precomputation time (in ms)

	Direct computation	Global memory	Constant memory
GeForce 32 bits	317.7	401.9	317.1
GeForce 64 bits	824.8	546.1	555.1
Quadro 32 bits	2437.7	2663.5	2526.1
Quadro 64 bits	4407.1	4290.3	4039.6

4.3 Kernel precomputation

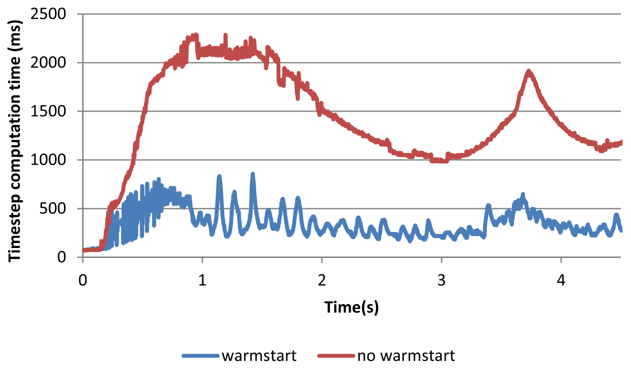
In this section, we study the usage of a precomputed lookup table for the kernel values as proposed by Bender et al. [5]. This optimization consists in sampling the kernel values at regular intervals and then using a lookup table to obtain the kernel values when required instead of using the actual kernel computation. In their paper, which uses a CPU multi-threaded implementation, this optimization allows for a reduction of the global simulation time of around 30% with 1000 sampling values. This optimization affects nearly every step of the DFSPH algorithm, except the neighbors search step, as it replaces the computation of W_{ij} and ∇W_{ij} by memory lookups (see Algorithms 1, 2 and 3). The main limitation of this optimization on a GPU implementation is that the cost of memory access relative to the cost of the direct computation is heavier on GPU than on CPU which may nullify the benefit of skipping the kernel computation. However, as noted in [9] the GeForce lineup of graphics card is only suited for single precision floating point computation. On such cards, double precision floating point computation may run as much as 32 times slower which will probably favor the use of a lookup table. On the Quadro card the double precision floating point computation is only 2 times slower than the 32 bits computation so the use of a lookup table may not be as advantageous. There are two ways to implement such system for GPU using CUDA. The lookup table can either be stored in global memory or in constant memory. The constant memory is supposed to allow better cache access which should give better performances. However, since the constant memory is limited to a maximum of 64KB or 128KB on recent architectures and even only 1000 sampling values would already use 8KB or 16KB depending on the chosen floating point precision. As mentioned earlier we use a cubic spline for our kernel as commonly done in SPH frameworks. As the cubic spline is relatively fast to compute, it may favor the direct computation while heavier kernels such as a quintic spline may lead to more advantageous results for the lookup table. In our experiments, we use 3 different configurations : direct kernel computation, lookup tables using global memory and lookup tables using constant memory. And similarly to previous works, we use 1000 sampling points.

The results are shown in Table 4. We can see that the results fit our expectations. On a GeForce 32 bits implementation, the use of the lookup table gives no reduction of the computation time and even increases it by close to 25% if we only use a global memory implementation. However, on the double floating point precision implementation, the use of the lookup table reduces the computation time by around 33% for both implementations. More surprisingly, the global memory implementation was slightly faster than the constant memory one with a 64 bits implementation every time we run this test. On the Quadro card, the impact of using a lookup table is less important. Similarly to the GeForce card, in the single floating point precision implementation, we also observe a raise of the computation time if a lookup table using global memory is used and no variation when using the constant memory. This time however, the increase is less noticeable (around 3%). As expected, the cost of using double floating point precision computation being lower on Quadro cards, the benefit of using the lookup table is also lower: 3% for the global memory implementation and 8% for the constant memory implementation.

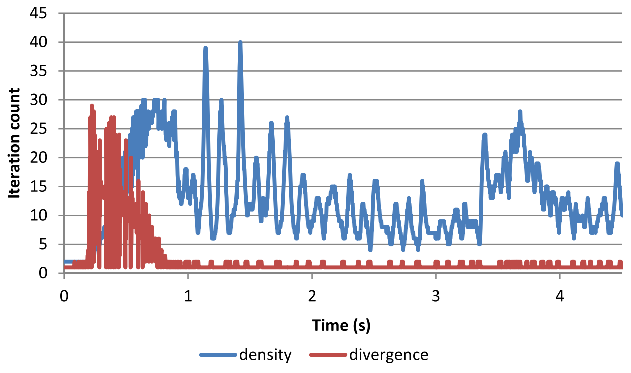
To conclude on this experiment, we would recommend to use a constant memory implementation if either double floating precision or both precision are used since it does not cost any computation time even if a single precision implementation is also used. However, if only the single precision is of interest it is possible to save some constant memory space with no computation cost by not using this optimization. Seeing as the cubic spline is one of the simplest kernels we can extrapolate our results by stating that if any more complex kernels are used it is then recommended to use the constant memory implementation regardless of the floating point precision.

5 DFSPH warm-start

In this experiment, we aim to study the impact of the warm-start mechanism commonly used with the DFSPH algorithm [6]. The warm-start mechanism is visible in the algorithms of both the constant density solver (see Algorithm 2, lines 3, 4 and 11) and the divergence-free solver (see Algorithm 3, lines 3, 4 and 11). To visualize the importance of warm-start, Fig. 4a shows the evolution over time of the computation time per time-step in a dam-break scenario. We can see the huge benefit brought by the warm-start resulting

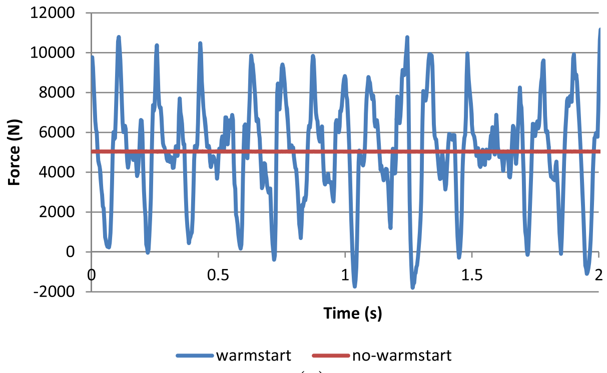


(a)

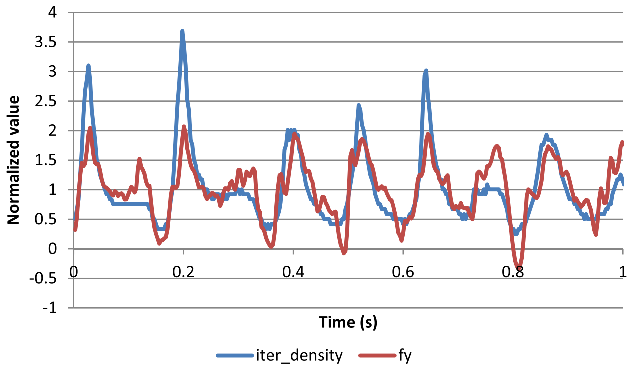


(b)

Fig. 4 **a** Computation time for each time-step during a dam-break with and without warm-start. **b** Iteration count over time for each solver with warm-start



(a)



(b)

Fig. 5 **a** Evolution of the vertical component of the force on a sphere immersed in a fluid at rest. **b** Normalized value of the force f_y and number of iterations of the divergence solver

in a global reduction of the computation time by a factor of 4. However, we can notice that the warm-start can introduce a certain level of instability in the duration of the time-steps. If we look at the evolution of the number of iterations used for the density and divergence solvers (see Fig. 4b), we can see that the variations in computation time are due to variations in the number of iterations used by both solvers. It is interesting to note that those oscillations are not just some noise as we can see that they have quite a large period (around 10 to 15 iterations). In the following section, we will try to evaluate the impact of those oscillations and see if we can reduce them by changing the DFSPH parameters.

5.1 Instability in warm-start

The oscillations in the simulation time are problematic for interactive applications. Indeed, each time the simulation spikes the inter-activeness of the simulation will drop resulting in an inconsistent user interaction. However, as even the shortest simulation step takes already more than $200ms$ for 1.1M particles, interactive applications may be limited. However, these oscillations create a cyclic compression–decompression motion in the fluid. The amplitude of that motion is small, around 10–20% of the radius of the particles but it still generates significant instabilities to the forces generated when a solid is interacting with the fluid. To illustrate that problem, we use another simulation where a sphere is entirely immersed in a fluid at rest and we will study the evolution of the sum of the forces applied by the fluid on the sphere for each simulation step. We are using a sphere with a radius of $0.5m$ making sure the solid particles used to discretize the sphere are placed tangent to the surface on the inside of the sphere so that the simulated volume is close to correct. The force shows similar levels of instability on all three dimensions but to simplify the visualization of our results we only report the vertical component of the force (f_y). Figure 5a shows a comparison of the evolution of the force f_y between implementations using warm-start or not. We can see that the instability has extreme consequences on the force applied by the fluid on the sphere. At some time-step, the error on the force is around 100% of the expected value. Also by normalizing the force and iteration count values we can observe a nearly perfect fitting of the evolution of both values (see Fig. 5b) illustrating their tight dependence. The close fitting of the instability with the iteration count leads us to believe that it is possible to improve the stability of the current warm-start method by manipulat-

Table 5 Results of the warm-start study for every tested configurations of the predictive–corrective algorithm

	No warm-start		Warm-start							
Density threshold	0.1	0.01	0.1	0.01	0.005	0.0025	0.001	0.0001	0.1	0.01
Divergence threshold	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.01	0.01
Avg computation time	170	1046	82	165	218	308	400	898	188	196
Standard deviation (F_y)	247	4	2569	2512	2408	2464	2445	2496	2527	655
Iteration density	12.20	100.00	2.88	11.80	17.40	24.79	35.58	100.00	3.18	12.93
Iteration divergence	1.00	1.00	1.31	1.08	1.08	1.08	1.08	1.08	10.65	1.89

	Warm-start									
Density threshold	0.001	0.0001	0.1	0.01	0.001	0.1	0.01	0.001	0.0001	0.0001
Divergence threshold	0.01	0.01	0.001	0.001	0.001	0.0001	0.0001	0.0001	0.001	0.0001
Avg computation time	850	904	553	296	669	1095	1057	1560	1045	1776
Standard deviation (F_y)	702	854	1161	67	24	1198	39	25	36	55
Iteration density	78.44	100.00	2.67	14.43	59.47	2.66	13.23	57.56	100.00	100.00
Iteration divergence	1.20	1.42	56.92	10.49	4.62	100.00	100.00	100.00	9.11	100.00

Table 6 Standard deviation of the vertical component of the force applied on a fully immersed sphere depending on the precision required for the density and divergence solvers

Density divergence	0.1	0.01	0.001	0.0001
0.1	2568	2512	2444	2495
0.01	2526	655	701	853
0.001	1161	67	24	36
0.0001	1197	38	25	55

ing high-level parameters of the DFSPH algorithm. These parameters would be the ones having the most impact on the number of iterations of the predictive–corrective solvers, such as the density and divergence thresholds (see Sect. 5.2) or the minimum number of iterations for each solver (see Sect. 5.3). The outcomes from such experiment are useful when the fluid simulation is part of a larger simulation, when the user only has access to the high-level parameters or when the user has no knowledge or access to the inner working of the fluid simulation.

5.2 Improving stability through precision

To study in detail the impact of the warm-start on the stability of the forces produced by the fluid, we studied the possibility of improving the stability when using the warm-start by increasing the required precision for the density solver or the divergence solver. The results are presented in Table 5. We first observe that trying to solve the problem of the variation of the number of iterations of the density solver by setting a stricter density target does not work. Indeed, we can see that even setting the target all the way to 0.0001% does not improve the stability even though each time-step required 100 iterations of the density solver with the force still presenting a standard deviation of 50% of its average value. This particular configuration illustrates very well the negative impact of the warm-start. The number of iterations of both solvers is the same as when the warm-start is not used but the level of stability of the force is very different. However, by modifying the desired precision of the divergence solver we can improve the stability. Table 6 shows the force stability for each combination of the two precision parameters. Our first observation is that even when using a precision of 0.0001% for both solvers (which requires 100 iterations for both solvers) it is still not possible to replicate the same level of stability observed when the warm-start is not used. This means that if a high level of stability is required for a simulation, i.e., where a fluid is interacting with solid objects, not using a warm-start, implemented in its current form, will produce better results than using it. However, if the computation time is as important as the stability, not using a warm-start is not a realistic option. Indeed, in Table 6, we can see that any combination using at least 0.01% for the density precision and 0.001% for the divergence precision allows the forces standard deviation to stay close to 1% or even below. In particular, that exact configuration offers simulation steps that are still 3 times faster than when not using any warm-start. However, those parameters only give good computation times because the fluid is at rest. We can see that even with our initial 0.1% divergence target, the divergence solver may require a large number of iterations at the start of a dam-break simulation (see Fig. 4b). During our experiments, we have observed that if there is a large amount of turbulence or a global compression of the fluid (e.g., when the fluid hits the opposite wall in a dam-break scenario), the number of iterations of the divergence solver will increase.

Table 7 Results for the study of the impact of forcing higher minimum iteration count for the divergence solver

	No warm-start			Warm-start							
Density threshold	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.01	0.01	0.01	0.01
Divergence threshold	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Min divergence iter	1	2	3	1	2	3	5	1	2	3	5
Avg computation time	170	173	182	82	101	113	134	165	214	223	240
Standard deviation (F_y)	247	4114	4012	2569	1525	2192	2192	2512	149	114	186
Iteration density	12.20	11.92	11.68	2.88	3.20	3.29	3.28	11.80	15.67	16.28	15.05
Iteration divergence	1.00	2.00	3.00	1.31	2.00	3.00	5.00	1.08	2.00	3.00	5.00

If we were to use an even more restrictive threshold, the simulation could require 100 iterations of the divergence solver which would greatly increase the computation time. The best solution to this problem would most likely be to use an adaptive threshold for each solver, with the threshold getting more restrictive when there is a low amount of motion in the fluid. However, we did not explore that solution yet.

5.3 Improving stability through iteration count

If we look once again at the evolution of the number of iterations of each solver during the dam-break simulation (see Fig. 4b), we can see that the divergence solver requires only one iteration for many simulation steps. One iteration is our minimum and is hard-coded in the solver. We can see in Table 5 that when the fluid is at rest only simulations with a threshold of 0.001% or lower use a significant number of iterations. Since it reduces the standard deviation to 1% while only using 10 iterations, it may be possible to represent a pseudo-adaptive threshold by increasing the minimum number of iterations of the divergence solver. The results of this approach are presented in Table 7. We can see that as expected increasing the minimum number of iterations greatly improves the stability of the forces. Using at least 2 divergence iterations reduces the standard deviation of the forces from 50% to 3% when using our initial 0.01% and 0.1% values and using at least 3 iterations reduces it to 2%. However, we can see that using a significantly higher minimum number of iterations does not bring any further improvement and may even reduce the stability. To illustrate the problem we ran two additional experiments. The first one consisted in using a density threshold of 0.1%. We observed that augmenting the minimum number of iterations brought lower benefits, reducing the standard deviation from 25% to 15% and using more iterations degraded the force stability. By repeating this experiment without using warm-start, we were able to see that any increase in the minimum number of iterations augments the standard deviation of the force. However, it is important to note that the observed instability seemed significantly different from the one observed when the warm-start is used. Indeed, as mentioned earlier the instability introduced by the warm-start is of relatively low frequency. For example with our initial 0.01% and 0.1% parameters, each oscillation takes between 15 and 30 simulation steps. The observed instability when not using warm-start is due to a very high-frequency noise. In these experiments, the force varies between its highest and lowest value at each simulation step.

Therefore, we would recommend, when using the current warm-start implementation, keeping the original thresholds of 0.01% and 0.1% while using a minimum of 3 iterations for the divergence solver instead of the minimum of 1 used in the SPlisHSPlasH library [5] when simulating solids interacting with a fluid. If the stability if more important than the execution time, it is probable that deactivating the warm-start is currently the best solution. However, if there is no solid in the simulation using a minimum of 1 iteration allows for better performance with no real downside since the instability is not large enough to generate noticeable artifacts on the scale of the entire fluid. A better solution would be to find the values for adaptive thresholds for both solvers. However, a better approach would be to investigate why the current warm-start approach introduces the instabilities.

6 Conclusion

In this paper, we have studied the impact of two general optimizations on a GPU implementation of a SPH algorithm. We showed that the use of a Morton curve for the sorting of the particle data is not necessarily the best choice despite its widespread use in recent SPH frameworks. In particular, we showed that in our implementation using a simple XYZ curve while sorting the data at nearly every simulation step is the best choice performance-wise. We also showed that the use of a lookup table to access the kernel function values can greatly reduce the computation time, particularly on double floating point precision implementation. This result echoes the results that were observed with multi-threaded CPU implementations in previous works. Finally, we showed that even though necessary to produce reasonable computation times, the use of a warm-start mechanism in predictive–corrective SPH algorithms, at least in its current conception, greatly decreases the stability of the fluid, especially when the fluid is interacting with solids. Although we showed that it is possible to reduce this instability to reasonable levels by increasing the minimum number of iterations, it is only a temporary solution that can still be used on systems where the fluid simulation is already fully integrated. We would encourage future works to study in detail the reason behind the instability brought by the warm-start. Similarly, further works are still needed to properly study the impact of the various space filling curves. First, as noted in [3] the use of the space filling curves can bring other advantages that a direct reduction of the computation times, such as a better compression rate. Secondly, our conclusions may be limited to our current implementation with our interleaved neighbors storage structure. Repetition of these experiments in other GPU implementations, like GPUSPH or DualSPHysics, would be required for any further conclusions. Finally, as we showed that the use of a precomputed space filling curve has a relatively low impact on the computation time, even in our naive unoptimized implementation, using more complex space filling curves might bring significant benefit to the simulation even if they cannot be procedurally generated. Finally, we would like to note that the studied optimizations are not specific to the mathematical discretization of the Navier–Stokes equations used in the DFSPH algorithm. Instead, they relate to the memory usage and the iterative schemes used for the pressure solver of the SPH algorithm. As such, the outcomes of our work should be applicable to any fluid simulation using similar iterative schemes such as the PCISPH [27], the IISPH [20] or the PBF [23] algorithms.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s00371-021-02379-w.

Declarations

Declarations The code used for our experiments can be found in the following repository: https://gitlab.liris.cnrs.fr/npronost/sph_dynamic_window.

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Akinci, N., Ihmsen, M., Akinci, G., Solenthaler, B., Teschner, M.: Versatile Rigid-fluid Coupling for Incompressible SPH. ACM Trans. Graph. **31**(4), 1–8 (2012). https://doi.org/10.1145/2185520.2185558
2. Antuono, M., Colagrossi, A., Marrone, S., Molteni, D.: Free-surface flows solved by means of SPH schemes with numerical diffusive terms. Comp. Phys Commun. **181**(3), 532–549 (2010). https://doi.org/10.1016/j.cpc.2009.11.002
3. Band, S., Gissler, C., Teschner, M.: Compressed Neighbour Lists for SPH. Comput. Graph. Forum. **39**(1), 531–542 (2020)
4. Baruffa, F., Iapichino, L., Hammer, N.J., Karakasis, V.(2017): Performance optimisation of smoothed particle hydrodynamics algorithms for multi/many-core architectures. In: 2017 International Conference on High Performance Computing & Simulation (HPCS) pp 381–388 . https://doi.org/10.1109/HPCS.2017.64 ArXiv: 1612.06090
5. Bender, J.(2017): SPlisHSPlasH is an open-source library for the physically-based simulation of fluids.: InteractiveComputerGraphics/SPlisHSPlasH . https://github.com/InteractiveComputerGraphics/SPlisHSPlasH. Publisher: Interactive Computer Graphics
6. Bender, J., Koschier, D.(2015): Divergence-free smoothed particle hydrodynamics. In: Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation - SCA '15, pp. 147–155. ACM Press . https://doi.org/10.1145/2786784.2786796. http://dl.acm.org/citation.cfm?doid=2786784.2786796. Event-place: Los Angeles, California
7. Bender, J., Koschier, D.: Divergence-Free SPH for incompressible and viscous fluids. IEEE Trans. Vis. Comput. Gr. **23**(3), 1193–1206 (2017). https://doi.org/10.1109/TVCG.2016.2578335
8. Bender, J., Kugelstadt, T., Weiler, M., Koschier, D.(2019): Volume Maps: an implicit boundary representation for SPH. In: Motion, Interaction and Games, MIG '19, pp. 1–10. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3359566.3360077
9. Bilotta, G., Zago, V., Hérault, A.(2019): Design and implementation of particle systems for meshfree methods with high performance. In: S. Chickerur (ed.) High Performance Parallel Computing. IntechOpen . https://doi.org/10.5772/intechopen.81755. https://www.intechopen.com/books/high-performance-parallel-computing/design-and-implementation-of-particle-systems-for-meshfree-methods-with-high-performance
10. Cummins, S.J., Rudman, M.: An SPH projection method. J. Comput. Phys. **152**(2), 584–607 (1999)
11. Domínguez, J.M., Crespo, A.J., Gómez-Gesteira, M.: Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. Comput. Phys. Commun. **184**(3), 617–627 (2013)
12. Durand, M., Raffin, B., Faure, F.(2012): A packed memory array to keep moving particles sorted. In: 9th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS), pp 69–77. The Eurographics Association . https://hal.inria.fr/hal-00762593
13. Goswami, P., Eliasson, A., Franzén, P.(2015): Implicit incompressible SPH on the GPU. In: VRIPHYS
14. Goswami, P., Schlegel, P., Solenthaler, B., Pajarola, R.(2010): Interactive SPH simulation and rendering on the GPU. In: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '10, pp. 55–64. Eurographics Association, Goslar Germany, Germany . http://dl.acm.org/citation.cfm?id=1921427.1921437
15. Harada, T., Koshizuka, S., Kawaguchi, Y.(2007): Smoothed particle hydrodynamics on GPUs. In: Computer Graphics International, 40: 63–70. SBC Petropolis
16. Haverkort, H.(2017): How many three-dimensional Hilbert curves are there? J. Comput. Geom. 8(1): 206-281. https://doi.org/10.20382/jocg.v8i1a10. https://jocg.org/index.php/jocg/article/view/3036
17. Hoetzlein, R.C., Devtech, G.(2014): Fast fixed-radius nearest neighbors: Interactive million-particle fluids . Presenters: _:n2083
18. Huang, K., Ruan, J., Zhao, Z., Li, C., Wang, C., Qin, H.: A general novel parallel framework for SPH-centric algorithms. Proceed.

ACM Comput. Gr. Int. Tech. **2**(1), 1–16 (2019). https://doi.org/10.1145/3321360
19. Ihmsen, M., Akinci, N., Becker, M., Teschner, M.: A parallel SPH implementation on multi-core CPUs. Comput. Gr. Forum. **30**(1), 99–112 (2011)
20. Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C., Teschner, M.: Implicit incompressible SPH. IEEE Trans. Vis. Comput. Gr. **20**(3), 426–435 (2014)
21. Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A., Teschner, M.(2014): SPH Fluids in Computer Graphics. Eurographics 2014 - State of the Art Reports p. 22 pages . https://doi.org/10.2312/EGST.20141034. http://diglib.eg.org/handle/10.2312/egst.20141034.021-042. Artwork Size: 22 pages Publisher: The Eurographics Association
22. Koschier, D., Bender, J., Solenthaler, B., Teschner, M.(2019): Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids. Eurographics 2019 - Tutorials p. 41 pages . https://doi.org/10.2312/egt.20191035. ArXiv: 2009.06944
23. Macklin, M., Müller, M.: Position based fluids. ACM Trans. Gr. (TOG) **32**(4), 1–12 (2013)
24. Monaghan, J.J.: Simulating free surface flows with SPH. J. Comput. Phys. **110**(2), 399–406 (1994)
25. Rustico, E., Bilotta, G., Herault, A., Del Negro, C., Gallo, G.: Advances in Multi-GPU smoothed particle hydrodynamics simulations. IEEE Trans. Parallel Distrib. Syst. **25**(1), 43–52 (2014)
26. Skilling, J.(2004): Programming the Hilbert curve. In: AIP Conference Proceedings, 707: 381–387. AIP, Jackson Hole, Wyoming (USA) . https://doi.org/10.1063/1.1751381. http://aip.scitation.org/doi/abs/10.1063/1.1751381. ISSN: 0094243X
27. Solenthaler, B., Pajarola, R.: Predictive-corrective incompressible SPH. ACM Trans. Gr. **28**(3), 1–6 (2009). https://doi.org/10.1145/1531326.1531346
28. Verma, K., Szewc, K., Wille, R.(2017): Advanced load balancing for SPH simulations on multi-GPU architectures. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), IEEE, pp 1–7, Waltham, MA . https://doi.org/10.1109/HPEC.2017.8091093. http://ieeexplore.ieee.org/document/8091093/
29. Weiler, M., Koschier, D., Brand, M., Bender, J.(2018): A physically consistent implicit viscosity solver for SPH fluids. In: Computer Graphics Forum, vol. 37, pp. 145–155. Wiley Online Library . Issue: 2
30. Winchenbach, R., Akhunov, R., Kolb, A.: Semi-analytic boundary handling below particle resolution for smoothed particle hydrodynamics. ACM Trans. Gr. (TOG) **39**(6), 1–17 (2020)
31. Winchenbach, R., Kolb, A.: Multi-level memory structures for simulating and rendering smoothed particle hydrodynamics. Comput. Gr. Forum **39**(6), 527–541 (2020). https://doi.org/10.1111/cgf.14090
32. Winkler, D., Rezavand, M., Rauch, W.: Neighbour lists for smoothed particle hydrodynamics on GPUs. Comput. Phys. Commun. **225**, 140–148 (2018). https://doi.org/10.1016/j.cpc.2017.12.014

Samuel Carensac is currently a research engineer at Ubisoft after being one at the University Claude Bernard Lyon 1, France. He received his Ph.D. degree in computer science from the University of Lyon (INSA Lyon), France, in 2019. His research interests include fluid simulation, physics-based animation, character animation and AI.

Nicolas Pronost is an assistant professor at the University Claude Bernard Lyon 1, France. He received his Ph.D. degree in computer science from the University Rennes 1, France, in 2006 after which he worked as a postdoctoral researcher at the Zhejiang University of Hangzhou, China, and the EPFL, Switzerland, and then as an assistant professor at the Utrecht University, The Netherlands. His research interests include physics-based animation, soft body deformation and musculoskeletal simulation.

Saïda Bouakaz received the Ph.D. degree from Joseph Fourier University, Grenoble, France. Currently, she is a Full Professor in the Department of Computer Science, Claude Bernard University Lyon1 France. Her research interests include computer vision and computer graphic. The current emphasis of her work is the recognition of human motion, gesture recognition and computer animation.