Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS (2015)
F. Jaillet, G. Zachmann, and F. Zara (Editors)

Implicit Incompressible SPH on the GPU

Prashant Goswami †, André Eliasson‡ & Pontus Franzén‡

Blekinge Institute of Technology, Sweden

Abstract

This paper presents CUDA-based parallelization of implicit incompressible SPH (IISPH) on the GPU. Along with the detailed exposition of our implementation, we analyze various components involved for their costs. We show that our CUDA version achieves near linear scaling with the number of particles and is faster than the multi-core parallelized IISPH on the CPU. We also present a basic comparison of IISPH with the standard SPH on GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing

1. Introduction

Smoothed particle hydrodynamics (SPH) has emerged as a powerful method to simulate fluid behaviour in a number of graphics applications. The possibility of enforcing computationally inexpensive incompressibility has been introduced in predictive-corrective incompressible SPH (PCISPH) [SP09] and improved upon in implicit incompressible SPH (IISPH) [ICS*14], and later in divergence-free SPH (DFSPH) [BK15] in the context of particle simulation to achieve more realistic fluid behaviour in recent years. The ability of the IISPH solver to simulate millions of particles together with employing larger time steps makes it promising for real-time considerations.

Similar to other particle-based methods, a large particle count is desirable in IISPH to achieve high resolution features in the simulation. To this end, sequential and multi-core simulations remain beyond the scope of any real-time purpose for more than a few thousand particles. Most modern graphics applications (games, simulators etc.) make massive use of the on-board computational power of the GPUs, where fluid simulation could be just one of the several components. Therefore, in order to fit efficiently within the existing framework, it is desirable to explore a GPU solution for IISPH.

We propose a completely parallel, GPU-based solution for IISPH using CUDA. The particles are resident on the GPU-memory throughout the simulation and could be used for rendering in the next step on the graphics card itself. We show that real-time to interactive frame rates are achievable using our simple solution, exhibiting linear dependence on the number of particles. We also give an initial estimate of how IISPH fairs in comparison to the standard, compressible SPH solver in terms of performance on the GPU.

2. Related Work

SPH has several benefits over other fluid simulation techniques like simplified boundary handling, obtaining fine-scale effects like splashes and implicit mass conservation. It was first introduced to computer graphics in [DG96] for deformable bodies and later to simulate fluids [MCG03]. Later [BT07] came up with a weakly compressible model for free surface flows.

Traditional SPH formulation needs a large stiffness value (and hence small time steps) to simulate incompressible behaviour, thereby increasing the computational time. The first promising work in this direction in computer graphics was by [SP09] where the density of particles are fixed in a predictive-corrective manner. The method was a significant improvement as it eliminated solving computationally expensive matrices for Poisson equation while still being able to handle large time steps.

Further work improved the state-of-the-art for simulating incompressible fluid behaviour, mostly towards reducing the cost. [RWT11] employ a hybrid approach by solving Poisson

† prashant.goswami@bth.se
‡ *the authors have equal contribution*

DOI: 10.2312/vriphys.20151331

equation on a coarse grid and transferring the initial pressure estimate to particles for density correction. In [HLL*12] Poisson equation is solved locally and integrated with the predictive-corrective framework. In [MM13] incompressibility is formulated as a positional constraint satisfying problem which enables taking even larger time steps than PCISPH. IISPH [ICS*14] improves the predictive-corrective incompressible SPH method by using a modified projection scheme which improves the convergence rate of the solver. Though multi-core CPU parallelization yields a linear parallel scaling as demonstrated in [TSG14], the frame rates are no longer interactive beyond a few thousand particles.

Several methods improve the efficiency of standard SPH. These include developing hybrid SPH-FLIP model to reduce actual physical particles in memory [CIPT14], using multiple resolutions [SG11], skipping computation on inactive fluid parts [GP11] and employing regional time stepping [GB14]. In this work, we focus our discussion on techniques more closely related to ours and refer the reader to [IOS*14] for an in-depth survey of the various fluid simulation methods.

Recent growth in the hardware capabilities has seen an emergence of efficient parallel solutions, particularly on the GPU. [HKK07,ZSP08] were one of the first to propose GPU solutions for SPH which employed shaders for the purpose. [GSSP10] came up with a CUDA-based solution utilizing shared memory, which scores both on the memory usage and efficiency fronts. Continuing along the line, the aim of this work is to explore the parallel GPU porting of IISPH.

3. IISPH Basics

Similar to PCISPH, IISPH does not rely on any stiffness parameter that appears in the equation of state (EOS) to compute pressure. However, it uses a discretization of PPE (pressure Poisson equation) which is a variant of earlier introduced incompressible SPH (ISPH) formulation in [ESE07]. The density (and error) prediction is obtained by discretized continuity equation.

IISPH solver comprises three major steps: advection prediction, pressure solving and moving particles, in that order. In the advection stage, density ρ and velocity $\mathbf{v}$ of all particles are predicted using external forces like viscosity and gravity. The particles then enter density correction iteration wherein the pressure value for each particle is updated which in turn updates its density. The iterative loop is designed to continue until all the particles have density error below a specified threshold η, subjected to a minimum of 2 iterations. Finally the velocity and position of the particles are obtained from the computed forces.

A major difference here with respect to PCISPH is that the density correcting pressure is built implicitly by iteratively solving linear equation for each particle and not directly computed and accumulated from the density error. The implicit formulation improves the convergence of the iterative solver. Similar to previous methods [SP09], [MM13], during a simulation step the computed neighbourhood set is reused through the iterations. The IISPH method is as outlined in Algorithm 1. We refer the reader to [ICS*14] for a more detailed understanding of the method.

Algorithm 1: IISPH Method

procedure PREDICT ADVECTION
  foreach *particle i* **do**
    compute $\rho_i(t) = \sum_j m_j W_{ij}(t)$
    predict $\mathbf{v}_i^{adv} = \mathbf{v}_i(t) + \Delta t \frac{\mathbf{F}_i^{adv}(t)}{m_i}$
    $\mathbf{d}_{ii} = \Delta t^2 \sum_j -\frac{m_j}{\rho_i^2} \nabla W_{ij}(t)$
  foreach *particle i* **do**
    $\rho_i^{adv} = \rho_i(t) + \Delta t \sum_j m_j (\mathbf{v}_{ij}^{adv}) \nabla W_{ij}(t)$
    $p_i^0 = 0.5 p_i(t - \Delta t)$
    compute a_{ii}
procedure PRESSURE SOLVE
  $l = 0$
  while $(\rho_{avg}^l - \rho_0 > \eta) \vee (l < 2)$ **do**
    foreach *particle i* **do**
      $\sum_j \mathbf{d}_{ij} p_j^l = \Delta t^2 \sum_j -\frac{m_j}{\rho_j^2(t)} \nabla W_{ij}(t)$
    foreach *particle i* **do**
      compute p_i^{l+1}
      $p_i(t) = p_i^{l+1}$
    $l = l + 1$
procedure INTEGRATION
  foreach *particle i* **do**
    $\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i^{adv} + \Delta t \frac{\mathbf{F}_i^p(t)}{m_i}$
    $\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t)$

4. IISPH on CUDA

Neighbourhood determination: Like other particle-based methods, the first step in IISPH is to determine neighbourhood set for each particle. To this end, we follow a similar approach as laid out in [GSSP10] with some changes. The simulation domain is divided into a virtual indexing grid along each axis, with block size equal to the support radius for the particles. However, we replace Morton based $z-index$ with simple linear index for all blocks in the simulation domain,(**Kernel** *calcIndex* in Algorithm 2). This avoids expensive bitwise operations while still maintaining coherence between adjacent particles. The particles are then sorted based on this index using fast radix sort provided by *thrust* library (**Kernel** *sortParticles*). To benefit from the increased memory on modern GPUs and in order to be able to take maximum advantage of available parallelism, for each particle we explicitly store the indices of all its neighbouring particles. The shared memory technique introduced in [GSSP10] could be still useful to fit higher particle count on a limited GPU memory, though with some penalty on parallelism.

The old particle data like positions and velocities (assigned to CUDA texture) are copied to new sorted index in the global memory array (**Kernel** *sortParticleData*). In order to determine the neighbours, the start and end index of each cell in the virtual grid is stored (**Kernel** *findGridCellStartAndEnd*). This is accomplished by simple linear scanning of sorted particles which update their index in the respective cell for the *start* and *end*. Finally, each particle collects its neighbours from the cell it lies in together with the adjoining 26 cells (**Kernel** *updateNeighbours*). The neighbouring indices are stored for each particle and are reused each time neighbours are fetched during subsequent computations in the iteration. The above operations are launched as separate kernels, see also Algorithm 2.

Predict advection: The PREDICT ADVECTION procedure comprises two separate kernels. In the first step, density is predicted which is required for the computation of factor $\mathbf{d}_{ii}$. $\mathbf{v}_i^{adv}$ is obtained using viscosity and external forces such as gravity. Thereafter, launch of a second kernel is necessitated by the observation that all particles need to finish with $\mathbf{d}_{ii}$ calculation before ρ_i^{adv} because of the relative velocity which appears in the density formulation. Following this, the factor a_{ii} is determined for each particle.

$$a_{ii} = \sum_j m_j(\mathbf{d}_{ii} - \mathbf{d}_{ji})\nabla W_{ij}$$

Whereas the value of $\mathbf{d}_{ii}$ computed from previous kernel is used, $\mathbf{d}_{ij}$ requires an extra computation (but no storage). Since the value of a_{ii} remains unchanged for an iteration, it is stored in the global memory for each particle and subsequently made a CUDA texture.

Pressure solve: The density is resolved through two or more corrective loops on each particle per iteration. For this purpose, the kernels in PRESSURE SOLVE are launched from the standard CPU call for each loop. **Kernel** *computeSumPressureMovement* updates $\sum_j \mathbf{d}_{ij} p_j^l$, which is required in **Kernel** *computePressure* to obtain the updated pressure p_i^{l+1}.

$$p_i^{l+1} = (1-\omega)p_i^l + \omega\frac{1}{a_{ii}}\Big(\rho_0 - \rho_i^{adv} - \sum_j m_j\Big(\sum_j \mathbf{d}_{ij}p_j^l - \mathbf{d}_{jj}p_j - \sum_{k\neq i}\mathbf{d}_{jk}p_k^l\Big)\nabla W_{ij}\Big)$$

The quantity $\sum_j \mathbf{d}_{ij} p_j^l$ is supplied as a CUDA texture since it is used first for the computation of p_i^{l+1} and then ρ_i^{l+1}. Furthermore, $\sum_{k\neq i}\mathbf{d}_{jk}p_k$ is also derived from it. At the end of each loop, the new density ρ_i^{l+1} (**Kernel** *computePredictedDensity*) and hence the error ρ_i^{err} (**Kernel** *computeDensityError*) is computed.

$$\rho_i^{l+1} = \rho_i^{adv} + p_i\sum_j m_j(\mathbf{d}_{ii} - \mathbf{d}_{ji})\nabla W_{ij} + \sum_j m_j\Big(\sum_j \mathbf{d}_{ij}p_j^l - \mathbf{d}_{jj}p_j^l - \sum_{k\neq i}\mathbf{d}_{jk}p_k^l\Big)\nabla W_{ij}$$

Kernel	Texture Attributes
computeDisplacementFactor	$\mathbf{x}_i, \mathbf{v}_i, N_i$
computeAdvectionFactor	$\mathbf{x}_i, \mathbf{v}_i^{adv}, \mathbf{d}_{ii}, N_i$
computeSumPressureMovement	$\mathbf{x}_i, \rho_i, p_i, N_i$
computePressure	$\mathbf{x}_i, \mathbf{d}_{ii}, \rho_i, \sum_j \mathbf{d}_{ij}p_j, N_i$
computePredictedDensity	$\mathbf{x}_i, \mathbf{d}_{ii}, \rho_i, \sum_j \mathbf{d}_{ij}p_j, N_i$
computeDensityError	ρ_i^{err}
calcIntegration	$\mathbf{v}_i^{adv}, \rho_i, p_i, N_i$

Table 1: Quantities supplied as texture for the various CUDA kernels.

CPU	Intel Core i7-3770 (3.4GHz)
GPU	MSI GeForce GTX 970 Gaming 4G
RAM	DDR3, 1600MHz, 16GB
OS	Windows 7 Ultimate 64-bit

(a) Setup 1

CPU	Intel Xeon E5-1650 (3.2GHz)
GPU	NVIDIA Quadro K4000
RAM	DDR3, 1600MHz, 16GB
OS	Windows 8.1 Enterprise 64-bit

(b) Setup 2

Table 2: Hardware specifications for experiments.

The total number of corrective loops depends on the density error. In order to estimate the global density error, we launch the reduction kernel in CUDA which supports parallel addition and maximum finding operations.

Even though dynamic parallelism is supported in recent versions of CUDA with compute capability 3.5 or higher, the dependence of quantities makes it infeasible to efficiently launch a kernel from within another kernel. This is because we require global synchronization as opposed to synchronization just within a block (for example, ρ_i^{l+1} depends upon the computation of its neighbours' p_j^l).

Integration: The new velocity is computed for each particle by summing up pressure forces from the neighbours and adding the contribution to the advection component $\mathbf{v}_i^{adv}$. For this the complete neighbour set is made available to each particle as texture memory.

The complete list of quantities supplied as texture memory for the various CUDA kernels is given in Table 1.

5. Results

The proposed approach was implemented in C++ using DirectX and HLSL shaders. We used two different setups for our experiments as given in Table 2.

In all scenarios a fixed time-step of 3.5 ms and particle spacing of 0.09 m was used. The density error threshold η was set to 1% of the rest density ρ_0 (= 1000). All the kernels were launched with a maximum of 256 threads per

Algorithm 2: Parallel IISPH on CUDA

procedure *NEIGHBOUR DETERMINATION*
  Kernel *calcIndex*
  Kernel *sortParticles*
  Kernel *sortParticleData*
  Kernel *findGridCellStartEnd*
  Kernel *updateNeighbours*
procedure *PREDICT ADVECTION*
  Kernel *computeDisplacementFactor*
    foreach *particle i* **do**
      compute $\rho_i(t)$
      predict $\mathbf{v}_i^{adv}$
      compute $\mathbf{d}_{ii}$
  Kernel *computeAdvectionFactor*
    foreach *particle i* **do**
      compute ρ_i^{adv}
      compute p_i^0
      compute a_{ii}
procedure *PRESSURE SOLVE*
  $l = 0$
  while $(\rho_{avg}^l - \rho_0 > \eta) \vee (l < 2)$ **do**
    Kernel *computeSumPressureMovement*
      foreach *particle i* **do**
        compute $\sum_j \mathbf{d}_{ij} p_j^l$
    Kernel *computePressure*
      foreach *particle i* **do**
        compute p_i^{l+1}
        $p_i(t) = p_i^{l+1}$
    Kernel *computePredictedDensity*
      foreach *particle i* **do**
        compute compute ρ_i^{l+1}
        compute ρ_i^{err}
    Kernel *computeDensityError*
      compute ρ_{avg}^l
    $l = l + 1$
procedure *INTEGRATION*
  Kernel *calcIntegration*
    foreach *particle i* **do**
      compute $\mathbf{v}_i(t + \Delta t)$
      compute $\mathbf{x}_i(t + \Delta t)$

block. All the associated constant variables were stored in the constant memory. DirectX 11.0 was used for rendering, billboards were used to represent each particle. Using the interoperability with DirectX, particles were always resident on the GPU memory and were never transferred back to the CPU. Real-time surface construction could be achieved on GPU with [Nvi15].

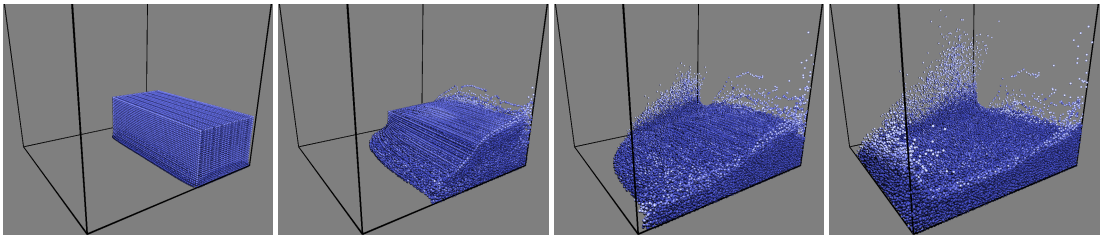
Three versions of the algorithm were implemented. The first one was a sequential version running on the CPU. The second one was the proposed parallel version running on the GPU. A simple parallel implementation of the CPU-version was also developed using OpenMP for comparison, though [TSG14] would give a better estimate on this front. The various scene set-ups used are as shown in Figure 1. The CUDA solution is able to achieve higher performance than an OpenMP implementation on the CPU. Setup 1 (Table 3) achieved a speed-up of about 6 times compared to the parallel CPU version. In setup 2 (Table 4) the speed-up (about 2 times) was lower because of a slower graphics card.

The CPU has a steady time per frame regardless of the scene while the CUDA times fluctuate slightly on computer setup 2 but on setup 1 the fluctuation is greater, see also Figure 2. Although the scenes differ slightly from each other, they still follow a linear growth when the number of particles increases. A similar linear growth in memory usage was detected in both the CPU- and the GPU-implementation for all test scenes, see Figure 3.

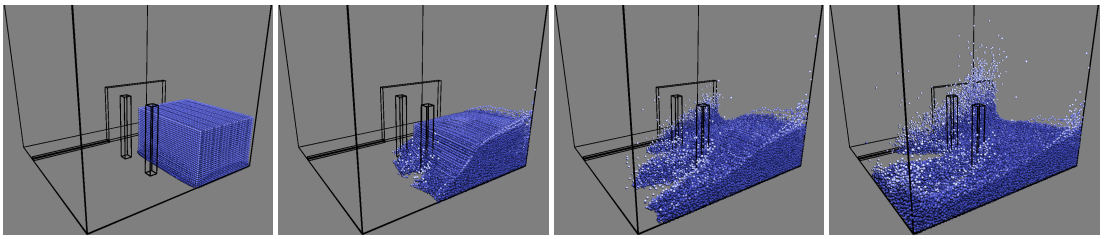
Figure 4 demonstrates the average time spent on each CUDA kernel in our implementation on the faster setup. For larger particle counts, **Kernel** *computeDisplacementFactor*, **Kernel** *computePredictedDensity*, **Kernel** *computePressure* and **Kernel** *computeAdvectionFactor* are the most expensive ones (in that order). Further, we notice that the neighborhood computation is not so expensive when compared to the other kernels. Using our implementation, we obtain a total occupancy of about 0.75 in our experiments.To estimate the cost of incompressibility with IISPH, the presented approach was applied to standard, compressible SPH implementation and measured with both setups. The results are as given in Table 5. A low stiffness constant of 1000 was employed and the time steps were computed using CFL condition. The results for both setups in Table 5 follow a linear growth rate just like IISPH. For $175K$ particles, standard SPH is around 2.3 times faster than IISPH in terms of the number of physics iterations executed per second.

6. Conclusions

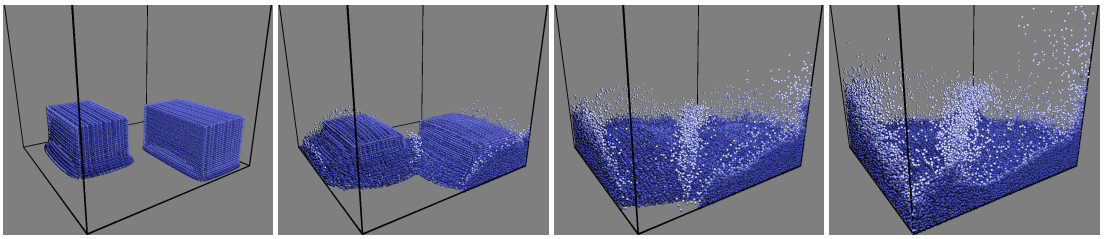
In this paper, we presented an efficient, CUDA-based parallel implementation of IISPH method. The proposed technique performs faster than the multi-core CPU-based parallel implementation and achieves near-linear scaling with the number of particles. A related future work would be to compare the GPU version of IISPH with other methods like divergence-free SPH and position-based fluids for efficiency.



(a) Simple: 121K particles



(b) Gallery: 99K particles



(c) Two blocks: 116K particles

Figure 1: Time lapse visualization of three different scenes with varying particle counts.

Setup 1	Physics - GPU		Physics - CPU (OpenMP)		
Particles	Time (ms)	FPS	Time (ms)	FPS	Speedup
7 600	2.58	388	6.89	146	2.67
20 000	4.06	247	22.18	45	5.47
54 000	10.21	100	64.20	16	6.29
103 000	21.07	49	126.80	8	6.02
175 000	39.18	28	221.16	5	5.64

Table 3: The average time and fps (frames per second) measured on setup 1 for each scene with a calculated speed-up between the GPU and parallel CPU.

Setup 2	Physics - GPU		Physics - CPU (OpenMP)		
Particles	Time (ms)	FPS	Time (ms)	FPS	Speedup
7 600	4.58	219	5.34	188	1.17
20 000	9.45	106	16.23	62	1.72
54 000	23.18	43	46.23	22	1.99
103 000	47.82	21	92.29	11	1.93
175 000	83.06	12	161.06	6	1.94

Table 4: The average time and fps measured on setup 2 for each scene with a calculated speed-up between the GPU and parallel CPU.

SPH CUDA	**Setup 1**		**Setup 2**	
Particles	time (ms)	FPS	time (ms)	FPS
7 600	0.79	1268	1.42	707
20 000	1.15	874	3.89	277
54 000	3.19	329	6.90	146
103 000	7.42	173	14.04	76
175 000	16.55	64	22.19	46

Table 5: Table showing the results of a standard SPH implementation on the two setups for comparison.

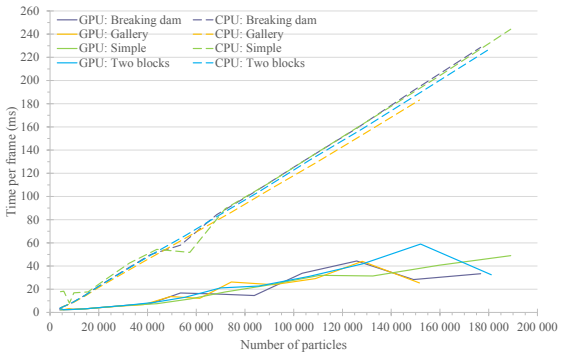
7. Acknowledgements

We thank anonymous reviewers for their constructive comments that helped us to improve the paper. We also thank Pierre-Luc Manteaux (INRIA-Grenoble) for the insightful discussions on IISPH.

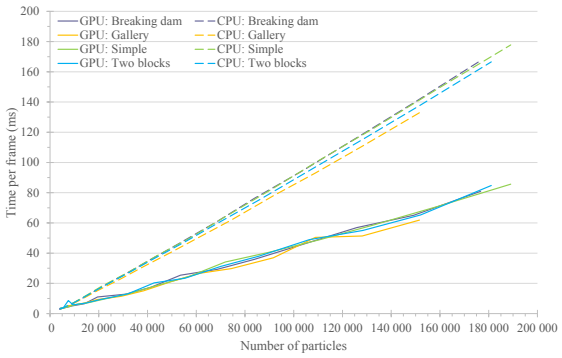
References

[BK15] BENDER J., KOSCHIER D.: Divergence-free smoothed particle hydrodynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2015), ACM. URL: http://dx.doi.org/10.1145/2786784.2786796. 1

[BT07] BECKER M., TESCHNER M.: Weakly compressible



(a) Setup 1



(b) Setup 2

Figure 2: The graph visualizes the computation time of the algorithm per frame on setup 1 (a) and setup 2 (b) for the CUDA-solution compared to OpenMP. All four tests scenes used a time-step of 3.5 ms and a particles spacing of 0.09 m. Measurements was taken over 1 000 frames.

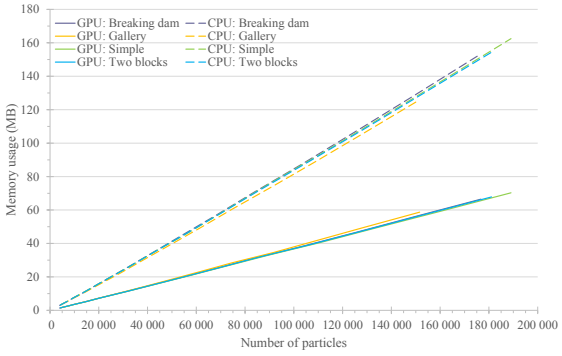


Figure 3: The GPU memory usage in all scenes grows linearly with the number of particles.

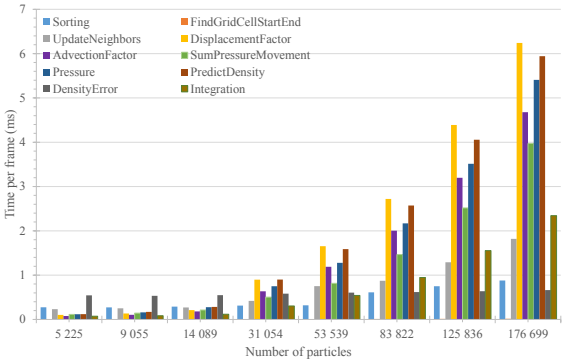


Figure 4: Split-up of the timings taken by the CUDA kernels in our GPU implementation.

SPH for free surface flows. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), SCA, Eurographics Association, pp. 209–217. URL: http://dl.acm.org/citation.cfm?id=1272690.1272719. 1

[CIPT14] CORNELIS J., IHMSEN M., PEER A., TESCHNER M.: IISPH-FLIP for incompressible fluids. *Computer Graphics Forum 33*, 2 (2014), 255–262. URL: http://dl.acm.org/citation.cfm?id=2771467. 2

[DG96] DESBRUN M., GASCUEL M.-P.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation* (New York, NY, USA, 1996), Springer-Verlag New York, Inc., pp. 61–76. URL: http://dl.acm.org/citation.cfm?id=274976.274981. 1

[ESE07] ELLERO M. B., SERRANO M., ESPAÑOL P.: Incompressible smoothed particle hydrodynamics. *Journal of Computational Physics* (2007). doi:10.1016/j.jcp.2007.06.019. 2

[GB14] GOSWAMI P., BATTY C.: Regional Time Stepping for SPH. In *Eurographics - Short Papers* (2014), Galin E., Wand M., (Eds.), The Eurographics Association. doi:10.2312/egsh.20141011. 2

[GP11] GOSWAMI P., PAJAROLA R.: Time Adaptive Approximate SPH. In *Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS"* (2011), Bender J., Erleben K., Galin E., (Eds.), The Eurographics Association. doi:10.2312/PE/vriphys/vriphys11/019-028. 2

[GSSP10] GOSWAMI P., SCHLEGEL P., SOLENTHALER B., PAJAROLA R.: Interactive sph simulation and rendering on the GPU. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2010), SCA, Eurographics Association, pp. 55–64. URL: http://dl.acm.org/citation.cfm?id=1921427.1921437. 2

[HKK07] HARADA T., KOSHIZUKA S., KAWAGUCHI Y.: Smoothed particle hydrodynamics on GPUs. In *Proc. of Computer Graphics International* (2007), pp. 63–70. URL: http://inf.ufrgs.br/cgi2007/cd_cgi/papers/harada.pdf. 2

[HLL*12] HE X., LIU N., LI S., WANG H., WANG G.: Local Poisson SPH For Viscous Incompressible Fluids. *Computer Graphics Forum* (2012). doi:10.1111/j.1467-8659.2012.03074.x. 2

[ICS*14] IHMSEN M., CORNELIS J., SOLENTHALER B., HORVATH C., TESCHNER M.: Implicit incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics 20*, 3 (Mar. 2014), 426–435. URL: http://dx.doi.org/10.1109/TVCG.2013.105, doi:10.1109/TVCG.2013.105. 1, 2

[IOS*14] IHMSEN M., ORTHMANN J., SOLENTHALER B., KOLB A., TESCHNER M.: SPH Fluids in Computer Graphics. In *Eurographics 2014 - State of the Art Reports* (2014), Lefebvre S., Spagnuolo M., (Eds.), The Eurographics Association. URL: https://diglib.eg.org/handle/10.2312/egst.20141034.021-042, doi:10.2312/egst.20141034. 2

[MCG03] MÜLLER M., CHARYPAR D., GROSS M.: Particle-based fluid simulation for interactive applications. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aire-la-Ville, Switzerland, Switzerland, 2003), SCA, Eurographics Association, pp. 154–159. URL: http://dl.acm.org/citation.cfm?id=846276.846298. 1

[MM13] MACKLIN M., MÜLLER M.: Position based fluids. *ACM Transactions on Graphics (TOG) 32*, 4 (2013), 104. 2

[Nvi15] NVIDIA Flex. URL: https://developer.nvidia.com/physx-flex. 4

[RWT11] RAVEENDRAN K., WOJTAN C., TURK G.: Hybrid smoothed particle hydrodynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (New York, NY, USA, 2011), SCA, ACM, pp. 33–42. URL: http://doi.acm.org/10.1145/2019406.2019411, doi:10.1145/2019406.2019411. 1

[SG11] SOLENTHALER B., GROSS M.: Two-scale particle simulation. *ACM Trans. Graph. 30*, 4 (July 2011), 81:1–81:8. URL: http://doi.acm.org/10.1145/2010324.1964976, doi:10.1145/2010324.1964976. 2

[SP09] SOLENTHALER B., PAJAROLA R.: Predictive-corrective incompressible SPH. *ACM Trans. Graph. 28*, 3 (2009), 40:1–40:6. URL: http://doi.acm.org/10.1145/1531326.1531346, doi:10.1145/1531326.1531346. 1, 2

[TSG14] THALER F., SOLENTHALER B., GROSS M. H.: A parallel architecture for IISPH fluids. In *VRIPHYS: 11th Workshop on Virtual Reality Interactions and Physical Simulations, Bremen, Germany, 2014. Proceedings* (2014), pp. 119–124. URL: http://dx.doi.org/10.2312/vriphys.20141230, doi:10.2312/vriphys.20141230. 2, 4

[ZSP08] ZHANG Y., SOLENTHALER B., PAJAROLA R.: Adaptive sampling and rendering of fluids on the GPU. In *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2008), SPBG, Eurographics Association, pp. 137–146. URL: http://dx.doi.org/10.2312/VG/VG-PBG08/137-146, doi:10.2312/VG/VG-PBG08/137-146. 2