Optimization of flexible neighbors lists in Smoothed Particle Hydrodynamics on GPU

Giuseppe Bilotta, Vito Zago, Alexis Hérault, Annalisa Cappello, Gaetana Ganci, Hendrik D van Ettinger, Robert A Dalrymple

Advances in Engineering Software 196 (2024) 103711

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Optimization of flexible neighbors lists in Smoothed Particle Hydrodynamics on GPU

Giuseppe Bilotta [a,*], Vito Zago [b], Alexis Hérault [a,c], Annalisa Cappello [a], Gaetana Ganci [a], Hendrik D. van Ettinger [d], Robert A. Dalrymple [b]

[a] Osservatorio Etneo, Istituto Nazionale di Geofisica e Vulcanologia, Catania, Italy
[b] Department of Civil and Environmental Engineering, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208, USA
[c] Laboratoire Modélisation mathématique et numerique, Conservatoire National des Arts et Métiers, 292 Rue Saint-Martin, Paris 75003, France
[d] Surf Loch, San Diego, CA, USA

ARTICLE INFO

Keywords:
GPUSPH
SPH
CUDA
Optimizations
Compilers
Neighbors list

ABSTRACT

Recent refactoring of the GPUSPH codebase have uncovered some of the limitations of the official CUDA compiler (`nvcc`) offered by NVIDIA when dealing with some C++ constructs, which has shed some new light on the relative importance of the neighbors list construction and traversal in SPH codes, presenting new possibility of optimization with surprising performance gains. We present our solution for high-performance neighbors list construction and traversal, and show that a 4× speedup can be achieved in industrial applications.

1. Introduction

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian, meshless numerical method for computational fluid dynamics originally created for astrophysics [1,2], and that has since grown to cover a wide range of fields [3] thanks to its ability to handle complex flows [4]. The Lagrangian, meshless nature of the method makes it particularly apt for free surface flows, violent flows, temperature-dependent fluids and non-Newtonian fluids [5–7].

One feature that makes the standard weakly-compressible form of SPH (WCSPH) particularly attractive from a computational point of view is the embarrassingly parallel nature of the method: the time evolution of each particle can be computed directly from the properties of the particle itself and those of its immediate neighborhood, without requiring the solution of any linear system, leading to straightforward implementation on massively parallel hardware.

In the last decades, graphic processing units (GPUs) have become a cheap alternatives to traditional CPU clusters as consumer-friendly parallel computing hardware [8,9]. The mass adoption of GPUs as computing solutions has been spearheaded by NVIDIA with CUDA, a runtime library with an associated single-source extension to the C++ programming language that makes it relatively easy to write software that can run on their GPUs [10].

GPUSPH was the first code to leverage the capabilities of CUDA with an implementation of WCSPH that could run entirely on NVIDIA GPUs [11], later followed by other open-source SPH codes with varying degrees of support for both CPU and GPU parallelization [12–14]. Throughout its history, performance has always been a priority in the development of GPUSPH, hence the choice to focus on a GPU-only implementation, and its expansion to multi-GPU [15] and multi-node (GPU clusters) [16] systems.

One of the key aspects of SPH as a Lagrangian meshless method is the neighbors list: this is an auxiliary data structure whose purpose is to reduce the computational complexity of the method from $O(N^2)$ (where N is the number of particles in a simulation) to $O(MN)$, where M is the (maximum) number of neighbors a particle interacts with, determined by the influence radius of the smoothing kernel that gives the method its name. The neighbors list construction is known to take a sizeable portion of the runtime of an SPH implementation, and a common strategy to reduce this impact is to avoid rebuilding it at every time-step, reducing the rebuilds with either a fixed frequency, or based on some measure of flow deformation, possibly in conjunction with larger search radii [17,18].

Following a refactoring of the GPUSPH codebase in version 5, aimed at improving the design and performance of the code [18,19] leveraging the growing support for more recent revisions of the C++ standard in CUDA, we have uncovered an interesting compiler-related bottleneck in our neighbors list construction and traversal codes that prevented it from scaling as expected on more recent GPU architectures.

* Corresponding author.
E-mail address: giuseppe.bilotta@ingv.it (G. Bilotta).

https://doi.org/10.1016/j.advengsoft.2024.103711
Received 29 September 2023; Received in revised form 11 June 2024; Accepted 17 June 2024
Available online 1 July 2024

The analysis and resolution of this bottleneck, that we present here, improves overall performance by a factor of 4, provides additional insights on the neighbors list management for SPH and other meshless methods running on GPU, and lays the ground for additional optimizations to be explored in the future.

This paper presents the details about how the issue was uncovered, the solution we have implemented to avoid it, and additional optimizations that have been explored as part of the process, including detailed performance comparisons.

Although some of the finer details are specific to the design adopted in GPUSPH, we believe that sharing our findings with researchers both within and outside of the SPH community will help drive a better understanding of the impact of the neighbors list management in SPH and other meshless methods, illustrate possible strategies to improve robustness and performance, underline that beyond algorithm and hardware improvements there is an untapped potential to improve performance purely by changing coding strategies, and show how exploring the wider compiler ecosystem can be a means to identify potentially problematic parts in scientific codes.

We will go into detail about some of the specifics of the GPUSPH implementation, including its use of some more advanced C++ features. While the paper is written to be as self-contained as possible, it does assume that the reader is already familiar with the fundamentals of C++ programming [20].

The paper starts with a brief introduction to the basics of WCSPH (Section 2), how these map to the features offered by GPUSPH, and how the development goals of the project affect their implementation (Section 3), with a particular focus on our split-neighbors strategy (Section 4) to provide all the background needed to follow the analysis and optimizations that constitute the core novelty of the work presented here.

In the main part of the paper we discuss the astonishing side effects of introducing support for alternative toolchains and how it helped identify additional bottlenecks in our code (Section 5), present the approaches we have adopted to resolve these bottlenecks, improving performance across all compilers (Section 6), and provide examples of the benefits this can bring to industrial applications of GPUSPH (Section 7). We discuss the implications of our results and draw our conclusions in Section 8

2. Weakly-compressible SPH

2.1. A lightning introduction to the method

As a Lagrangian method for computational fluid dynamics, weakly-compressible SPH is designed to solve the equations for the continuity of mass

$$\frac{D\rho}{Dt} = -\rho\nabla\cdot\mathbf{u} \tag{1}$$

and momentum (Navier–Stokes equations)

$$\frac{D\mathbf{u}}{Dt} = -\frac{\nabla P}{\rho} + \frac{1}{\rho}\nabla\cdot(\mu\nabla\mathbf{u}) + \mathbf{g} \tag{2}$$

where ρ represents the density, $\mathbf{u}$ the velocity, P the pressure, μ the dynamic viscosity, $\mathbf{g}$ the external body forces (e.g. gravity), and D/Dt is the Lagrangian (total) derivative with respect to time.

The system of equations is closed by an equation of states that relates the pressure P to the density ρ, typically Cole's [21,22] equation of state

$$P(\rho) = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{3}$$

where ρ_0 is the at-rest density for the fluid, γ the polytropic constant, and B a coefficient related to the at-rest sound speed c_0 by $B = \rho_0 c_0^2/\gamma$. Weak compressibility is achieved under the assumption that $c_0 > 10U$, where U is the maximum flow velocity, which ensures that relative density variations will remain below 1%.

Although the physical speed of sound of the fluid would be sufficient to guarantee the weak-compressibility condition in many applications with subsonic flows (Mach number <0.1), the spatial discretization of WCSPH (that will be presented momentarily) is often paired with an explicit integration scheme, for which the physical speed of sound would result in prohibitively small time-steps. In practical applications of WCSPH a fictitious sound speed is usually preferred, chosen lower than the physical one, but high enough to maintain the weakly-compressible regime.

In this case, in the computation of U (and thus c_0), one should take into account not only the actual velocity experienced by the particles due to the dynamics, but also the hydrostatic condition, defined by the theoretical free-fall velocity experienced by a particle dropping from the maximum fluid height to the lowest point: assuming g is the magnitude of $\mathbf{g}$ and H is the maximum distance that can be traveled by a particle in the direction of $\mathbf{g}$, the hydrostatic condition can be computed as $\sqrt{2gH}$.

With SPH, the computational domain Ω is discretized by a set of particles that act as interpolation nodes, but are free to move with respect to each other. Any field f is then discretized by representing it as a convolution with Dirac's δ distribution $f(\mathbf{x}) = \int_\Omega f(\mathbf{y})\delta(\mathbf{y}-\mathbf{x})d\mathbf{y}$, approximating Dirac's distribution by means of a family of *smoothing kernels* $W(\mathbf{r}, h)$ parametrized by the *smoothing length* h in such a way that $\lim_{h\to 0} W = \delta$ in the sense of distributions, and finally discretizing the integral as a summation over all the particles:

$$f(\mathbf{x}) \simeq \sum_j f(\mathbf{x}_j)W(\mathbf{x}_j - \mathbf{x}, h)V_j. \tag{4}$$

where V_i represents the volume of particle i, and is frequently expressed in terms of its mass and density as $V_i = m_i/\rho_i$.

The smoothing kernel is usually chosen radial (i.e. depending only on $r = \|\mathbf{r}\|$), with compact support (specifically, there exists $k > 0$ such that $W(\mathbf{r}, h) = 0 \forall \mathbf{r}$ s.t. $r > kh$) and unitary (i.e. such that $\int_\Omega W(\mathbf{r}, h)d\mathbf{r} = 1$).

The compact support implies that the summation (4) only extends to the *neighborhood* of $\mathbf{x}$ of radius kh (called the *influence radius* of the kernel). The radial symmetry implies that $W(\mathbf{r}, h) = \omega(r, h)$ for some function ω, and that the kernel gradient can be written as $\nabla W(\mathbf{r}, h) = \mathbf{r}F(r, h)$ where $F(r, h) \triangleq (1/r)\partial\omega/\partial r$, which is particularly convenient when F can be written analytically without an explicit division by r, improving numerical stability when r may become vanishingly small.

Using the standard SPH notation $\mathbf{x}_{ij} = \mathbf{x}_i - \mathbf{x}_j$, $W_{ij} = W(\mathbf{x}_{ij}, h)$, $F_{ij} = F(\left|\mathbf{x}_{ij}\right|, h)$, and $f_{ij} = f(\mathbf{x}_i) - f(\mathbf{x}_j)$ for any other field f, the SPH discretization of the gradient of a field f at the position of particle j which is far from the boundary of the domain can be written as

$$\nabla f(\mathbf{x}_i) \simeq \sum_j f(\mathbf{x}_j)F_{ij}V_j\mathbf{x}_{ij} \tag{5}$$

although symmetrized expressions, obtained by adding/subtracting the gradient of a constant field, are preferred, leading to expressions such as

$$\nabla f(\mathbf{x}_i) \simeq \sum_j \left(f(\mathbf{x}_i) + f(\mathbf{x}_j)\right) F_{ij}V_j\mathbf{x}_{ij} \tag{6}$$

or

$$\nabla f(\mathbf{x}_i) \simeq \sum_j f_{ij}F_{ij}V_j\mathbf{x}_{ij}. \tag{7}$$

The choice of the form for the discretization of the gradient leads to a variety of different SPH formulations [3,23–25]. For example, a common formulation, following Monaghan's "golden rule", expresses Eqs. (1) and (2) in discrete form as

$$\frac{D\rho_i}{Dt} = -\sum_j m_j\mathbf{u}_{ij}\cdot\mathbf{x}_{ij}F_{ij} \tag{8}$$

$$\frac{D\mathbf{u}_i}{Dt} = -\sum_j \left(\left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\right)\mathbf{x}_{ij} + 2K\frac{m_j}{\rho_i\rho_j}\bar{\mu}_{ij}\frac{\mathbf{x}_{ij}\cdot\mathbf{u}_{ij}}{\mathbf{x}_{ij}^2}\mathbf{x}_{ij}\right)F_{ij} + \mathbf{g} \tag{9}$$

while an alternative formulation taking ideas from Landrini [25] and Morris [26] gives:

$$\frac{D\rho_i}{Dt} = -\sum_j \frac{\rho_i}{\rho_j} m_j \mathbf{u}_{ij} \cdot \mathbf{x}_{ij} F_{ij} \tag{10}$$

$$\frac{D\mathbf{u}_i}{Dt} = -\sum_j \left(\frac{P_i + P_j}{\rho_i \rho_j} \mathbf{x}_{ij} + 2 \frac{\rho_i \rho_j}{\rho_i + \rho_j} \bar{\mu}_{ij} \mathbf{u}_{ij} \right) F_{ij} + \mathbf{g} \tag{11}$$

Additionally, dissipative terms may be added to both the momentum [3] and mass [27,28] conservation equations, to smooth out numerical noise. The expression for the smoothing kernel and its gradient can also be replaced by corrected versions that improve the consistency and/or conservative properties of the discretized operators [29–32].

Near the boundaries, the smoothing kernel support is incomplete and requires special treatment, and the treatment of boundary conditions remains one of the Grand Challenges in SPH [33]. For solid boundaries, several approaches have been proposed, including repulsive boundaries [3,34,35], static fluid-like particles [36,37], "dummy" boundary particles with properties interpolated from the neighboring fluids [38], semi-analytical approaches [39–41], and ghost particle methods [42], each with different degrees of accuracy, stability, and computational complexity.

2.2. From theory to implementation

Since the time derivatives of the particle properties in WCSPH can be computed from the properties of the particle and its neighbors, the method lends itself naturally to parallelization, especially when coupled with an explicit integration scheme.

On stream processing hardware such as GPUs, there is a natural mapping between work-items and particles that has been exploited for SPH implementation even before the birth of GPGPU-enabled hardware [9]. With this focus, we can talk about the *central particle* (the one being processed by the work-item), and its *neighbors*, the particles contained in the influence sphere of the central particle and that participate in the summations on the right-hand side of Eqs. (8)–(11) (or any other discretized formulation of choice).

The main iteration of most numerical implementations of WCSPH thus follows a scheme like the following:

neighbors search used to identify the neighbors of each particle;

computation of time derivatives computing $D\rho/Dt$, $D\mathbf{u}/Dt$, etc for each particle;

integration computing the new density, velocity and position.

The computation of time derivatives and their integration may happen multiple times per integration step, depending on the adopted scheme (e.g. once for a simple forward Euler integration scheme, twice in a predictor/corrector scheme, four times in a Runge–Kutta RK4 scheme). Additional steps may be necessary in special cases, too. For example, density diffusion terms may be applied after the integration step in certain formulations [43] or boundary conditions may need to be computed by extrapolating SPH-averaged values from the fluid to the boundary [26,38], or it may be necessary to compute the apparent viscosity before the forces computation when modeling non-Newtonian fluids [7,44].

On stream processing hardware, each of these steps will be enshrined in one or more *computational kernels*, functions associated with the central particle, and parallelized over the entire system according to the hardware capabilities. In most cases, the ideal storage system for the particle properties themselves (position, velocity, mass, density, apparent viscosity etc.) is that of a *structure of arrays*, where an individual array is used for each property, optionally merging some scalar and vector properties that are frequently used together: for example, in GPUSPH we use a single 4-component vector data type to store 3D position and mass, and another 4-component vector to store velocity and density. This is especially convenient for hardware such as GPUs, but is actually useful on most modern CPU systems as well [18], as it tends to naturally map array elements to the hardware vector types. Conversely, particle data that requires more than 4 components (e.g. symmetric tensors that require 6 components in 3D) may be inefficient to access on GPUs; in this case it may be convenient to split the storage into smaller units, such as a 4-component vector and a 2-component vector, or 3 2-component vectors, as illustrated in Fig. 1.

2.3. Optimizing the neighbors search

Each of the steps (with the possible exception of the integration steps) requires one or more loops (usually for summations) over the neighbors, pushing the need for an efficient neighbors search. The main strategy to improve the neighbors search performance is to adopt auxiliary data structures that help restrict the search space. Space partitions using trees are more common in applications where the smoothing length is variable: for example, n-trees (quadtrees in two dimensions, octrees in three) to bucket particles that are close in space are common in astrophysics, where they can be also used in support for gravity computation [45,46]. A brief review of tree-based methods, with some proposed enhancements, can be found in [47].

A straightforward auxiliary data structure that is very practical in case of fixed smoothing lengths and that also has additional uses is a simple grid of cells with side length no less than the influence radius (Fig. 1). The grid itself is represented simply by its origin (coordinates in 2D or 3D spaces), extents (dimensions in each coordinate direction), and grid spacing (which may be different in each coordinate direction). If particles are sorted in memory by the cell they belong to, bucketing can be achieved simply by storing the indices of the first and last particle in each cell. The particle sorting also brings performance benefits related to the improved data locality [48].

The primary objective of this auxiliary grid is to limit the neighbors search to the 9 (in 2D) or 27 (in 3D) cells around the central particle cell, but the same data structure is also useful to implement efficient multi-GPU and multi-node support [15,16] and uniform accuracy in space without the need for extended precision [49,50], by relying on local (cell-relative) particle positions.

Even with this improvements, the neighbors search can still be an expensive procedure, due also to the cost of maintaining the auxiliary data structures themselves, and/or to the particle sort. These costs however can be amortized (at the expense of memory consumption) by building a *neighbors list* per particle, to be rebuilt periodically [17]. The frequency at which the neighbors list needs to be rebuilt depends on the dynamics of the problem, and can be reduced by using a slightly larger radius for neighbors search compared to the actual influence radius. As we shall see in the upcoming sections, the way the neighbors list is stored, processed and built are essential details with a significant influence on the computational performance of an SPH implementation.

3. GPUSPH features

3.1. The SPH in GPUSPH

Born with the design goal of offering a simple, high-performance implementation of classic WCSPH [11], later extended with the aim to model lava flows [7,51], GPUSPH has since grown into a very sophisticated engine for SPH, with the ultimate objective of becoming a *universal SPH computational engine*, useful both for applications and for research in the numerical method itself.

This objective has driven recent development of GPUSPH to cover a growing swath of computational fluid dynamics for simple and complex fluids, as well as a vast collection of SPH formulations, all while striving for top performance, robustness and correctness. Satisfying

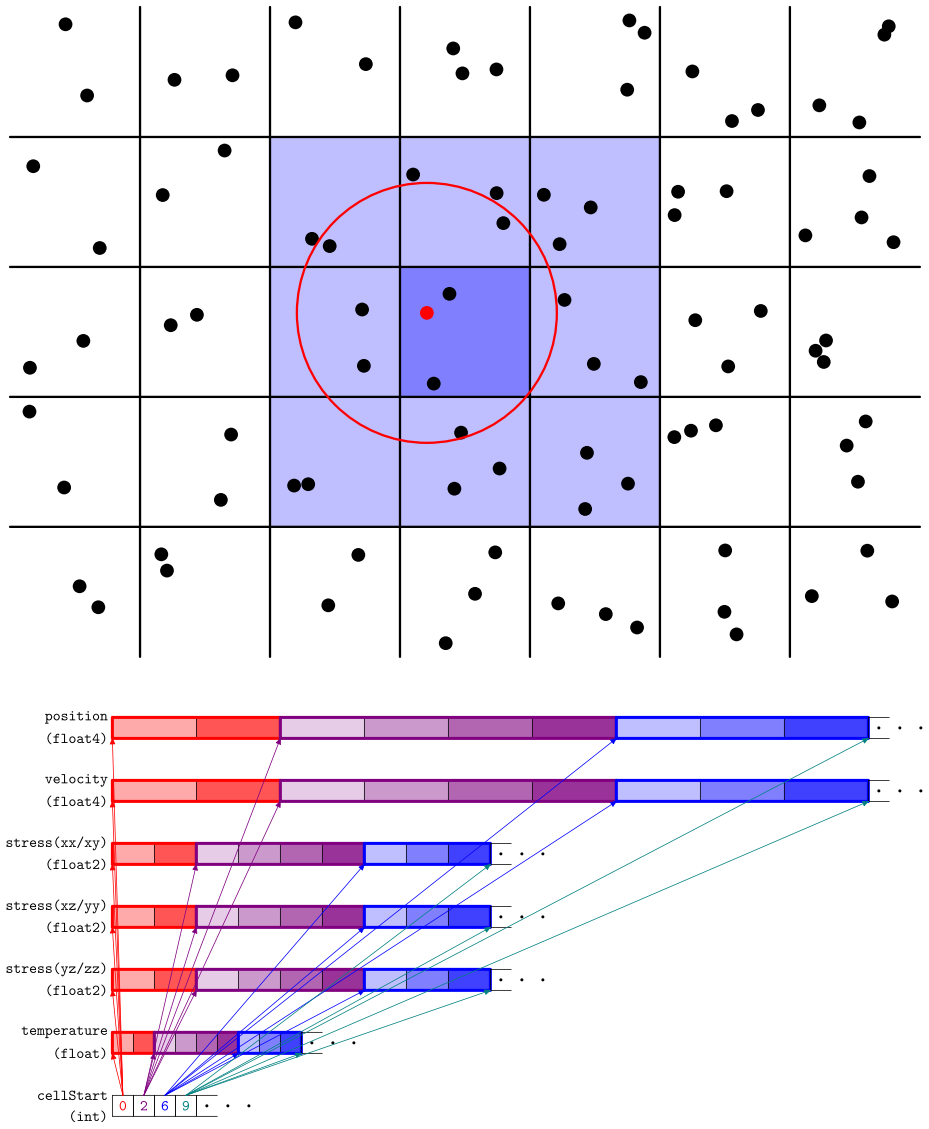

Fig. 1. Grouping particles by reference cells with a cell-side which is not less than the neighbors search radius allows the search to be limited to the particles in the Moore neighborhood of the cell to which the central particle belongs (top). This requires a sorting process where the actual particle data is moved in memory so that the particles in the same cell are located next to each other. An additional array can be used to track the offset in memory where the data for the particles in each cell begins (bottom).

these goals concurrently poses a significant coding challenge, especially in consideration of the vastness of the problem.

The SPH core of GPUSPH is the so-called "simulation framework", a collection of computational kernels specialized on the basis of user-selectable options that determine every aspect of the simulation: the equations to be solved (e.g. Navier–Stokes, heat, both), physical aspects such as the rheological or turbulence model, and choices about the details of the numerical model, such as the SPH formulation, the solid boundary model, the smoothing kernel, etc.

The comprehensiveness of the framework can be remarked by looking at the variety of options offered, summarized in Table 1 for the latest publicly released version. The total number of possible theoretical combinations, considering all options, is between 10^9 and 10^{10}, and that is before including additional options that have not been fully integrated in the public versions, such as coupling with the heat equation [7,65], coupling with finite-element models [66], or kernel gradient corrections [32,67]. Even though not all combinations are currently supported, the ultimate objective remains to cover the widest possible range. Still, to satisfy our goals of universality for both research and applications, this must be achieved *without any performance penalty for unused options*, and *minimizing implementation complexity*.

The lack of performance penalty for unused options is essential for the usefulness of GPUSPH in applications, and translates into the following maxim: the runtime of the program when a given framework option is not used/enabled should be the same as if support for that option had not been implemented in the code at all. This also means that when adding new features, there should be no regression in the runtime of any of the pre-existing options. This can be achieved by doing as much work as possible at compile time, and helping the compiler in producing optimal code by isolating code and variables that are specific to an option. Some of the details about how this is achieved in GPUSPH were discussed in [18,19]. A more in-depth presentation of these technical computing aspects will be the focus of a separate paper. We will only discuss here how this design choices have affected the handling of the neighbors list, which is the main locus of the optimizations presented in this paper.

3.2. The GPU in GPUSPH

GPUSPH was designed from the ground up to rely exclusively on GPUs for the computational part [11]. To achieve this, the program

Table 1
A summary of the framework options that can be set by the user when defining a test case in GPUSPH, with relevant bibliographical references. Not all combinations are supported. Some features are experimental and have not been fully merged into the public version yet.

Dimensionality	1D, 2D, 3D
Smoothing kernel	Quadratic, Cubic spline, Wendland, Gaussian
SPH formulation	WCSPH single-fluid [3], WCSPH multi-fluid [25], Grenier [23], Hu & Adams [24]
Density diffusion	(none), Ferrari [52,53], Brezzi [43,54], Molteni & Colagrossi [27], Antuono (δ-SPH) [28]
Boundary model	Lennard-Jones [3,34], Monaghan–Kajtar [35], dynamic [36,37], dummy [38], semi-analytical [39,40]
Periodicity	(any combination of) X, Y, Z
Rheological models	Inviscid, Newton, Granular [44], Bingham [55], Papanastasiou [56], Power-law [57], Herschel–Bulkley [58], Alexandrou [59], DeKee & Turcotte [60], Zhu [61]
Turbulence model	(none), k-ϵ [62], Sub-Particle Scale (SPS) [63], artificial viscosity (improperly) [3]
Viscous model	Morris [26], Monaghan [3], Español & Revenga [64]
Averaging operator	Arithmetic, harmonic, geometric
Internal viscosity representation	Dynamic Kinematic
Miscellanea (boolean flags)	Adaptive time-stepping XSPH Geometric planes support Geometric natural topography support Moving bodies support Open boundaries support Water depth computation Density summation Semi-analytical gamma quadrature Internal energy computation Multi-fluid support Repacking Implicit integration of the viscous term

execution is split into three phases: initialization, simulation, and data storage.

The initialization phase runs on program startup, on the host CPU, and it takes care of generating the initial particle distribution, either from a geometric description of the domain or from data stored on disk (e.g. when resuming an interrupted simulation).

After initialization, the worker threads are created, instantiating a `GPUWorker` for each GPU selected for the simulation, and the domain particles are distributed to the GPUs (i.e. the first and last particle assigned to each GPU is computed). Each worker then allocates data arrays on its GPU, forming the global buffer lists, and the buffer contents are initialized by copying data over from the host, for the subset of particles assigned to the specific device.

No further data exchange between the GPU and the host happens, except for the following circumstances:

- after the particle sorting and neighbors list construction, the host fetches information about the current number of particles, and the maximum number of neighbors per particle; this information is used to check if particles have been removed because they had gone out of bounds, to track the new distribution of particles in multi-GPU when particles cross from one device to an adjacent one, and to check that all neighbors could be accounted for (issues with the number of neighbors typically indicate either an incorrect initial particle distribution, or some issues with the choice of formulation or its implementation);
- the maximum allowed time-step (minimum over all the particles) is computed on device using a parallel reduction, and then downloaded to the host, for time-keeping;
- for fluid/structure interaction, the cumulative forces and torques exerted by the fluid on each rigid body are computed on the GPU, and downloaded to the host, to be passed to Project Chrono to compute the motion of the rigid body; the updated position of the center of mass and rotation of the rigid bodies is then copied back to the GPU, where the information is used to move the body particles accordingly.

For multi-GPU (both single- and multi-node) simulations, data is transferred directly from device to device if possible, i.e. if the GPUs can access each other's memory either through peering (on one machine) or through GPUDirect in multi-node configurations where the network setup supports it [15]. When this is not possible, data transfer happens through a staging area on host, which can negatively affect performance due to the additional memory copies.

Multi-GPU data transfers are explicitly marked by the integrator, allowing the developer to choose when to transfer data, and which data to transfer. These choices can be tuned to improve scaling by overlapping computations and data transfer [15,16].

Finally, to allow data storage to disk, all particle data arrays get downloaded to the host at fixed (simulated) time interval selected by the user. The simulation is suspended during this process.

It should be noted also that the arrays are only allocated once during the initialization phase. If the number of particles decreases during the simulation, e.g. because some particles fly out of the computational domain, the contents of the arrays are compacted during sorting and the additional entries are simply ignored. Allocations are made taking into account the possibility of the number of particles increasing because of open boundaries or in the multi-GPU case; in the open boundary case the user has control on the maximum number of particles that may be considered for the simulation, and in case of overflow the program terminates, allowing the user to resume with a higher upper bound. This avoids expensive reallocations at runtime.

4. The split neighbors list

One of the most significant performance boost from version 4 to version 5 of GPUSPH is the split neighbors list processing, which has brought a typical performance improvement between 15% and 30%, depending on the combination of framework options and hardware capabilities [18,19].

The idea for this strategy emerged from an analysis of the performance of the `forces` computational kernel in version 4 which revealed that despite the high density of computational operations in the kernel, its runtime was still largely memory-bound. The main cause for this was tracked down to the large number of variables that had to be allocated during the processing of the particle–particle interactions, which resulted in them overflowing the register banks of the GPU multiprocessors, resulting in the usage of the much slower VRAM as temporary storage (termed *local memory* in CUDA).

This large register pressure was ascribed to the monolithic nature of the kernel and the disparity of behavior in the interaction between particles of different types: indeed, for most boundary models fluid–fluid particle interactions are different from fluid–boundary particle interactions, but since particles are stored all together, and the neighbors list also stored all neighbors together (without distinction of type), during execution of the monolithic kernel each work-item would load the data for the particle being processed, and then traverse the entire neighbors list, deciding *at runtime* how to interact with each specific neighbor, based on the neighbor type.

This led to a growth in register usage, since the total number of variables that needed to be allocated was no less than the *union* of the variables needed for each of the particle–particle interaction types. This approach also led to additional performance loss due to the runtime decision about the kind of interaction, and the possible divergence of execution code-paths due to disparity of interaction between pairs of particles being processed concurrently, a well-known performance issue on GPUs.

The solution we adopted for version 5 was to split the `forces` kernel into a *separate* specialization for each of the particle–particle interaction pairs: one for fluid to fluid, one for fluid to boundary, one for boundary to fluid, etc. The split has been particularly meaningful for the semi-analytical boundary conditions [39,40], that have three different particle types, larger disparities in the treatment of different pairs, and complex rules to decide which pairs should be computed: even just moving most of the decision logic outside of the device-side computational kernel to the host-side has given a measurable performance gain of a few percents.

Reducing the complexity of the logic inside each instance of the computational kernel (now one per pairwise combination of particle types) gives more optimization opportunities to the compiler, especially when most of the conditions end up depending only on the compile-time parameters that define the framework options chosen by the user. On the other hand, by itself this strategy is insufficient, since leaving the kernel body and associated data structures unchanged still leaves some inefficiencies, one of which is relevant to our discussion: when all neighbors are stored together, it will be necessary to randomly skip elements from the neighbors list during its traversal. This can be quite costly, as it a likely source of divergence at the hardware level and leads to work-items fetching neighbors indices in an inefficient, scattered pattern. We have solved the issue by redesigning the way neighbors are stored in the list, in such a way that all neighbors of the same type are stored consecutively, while preserving GPU-optimal access patterns and without increasing the storage requirements.

4.1. Efficient split neighbors list

Memory access is a significant bottleneck on most modern computational hardware, be it CPUs or GPUs. This is the main reason for the growing, multi-layer caches of CPUs, and the introduction of L1 and L2 caches even on GPUs. Optimal memory access patterns can significantly speed up an implementation, but the optimality of the patterns depends not only on the nature of the algorithm, but also on the characteristics of the hardware.

On a mostly sequential processor like a CPU, the best cache usage is obtained by placing the data needed by a single thread in adjacent memory locations. By contrast, on stream processing hardware like GPUs, optimal access patterns require that adjacent memory locations refer to data needed concurrently by different work-items.

Let us consider the case of the neighbors list, and let us denote by $n_i(j)$ the index of the ith neighbor of the jth particle. When traversing the neighbors list, a thread or work-item processing particle j will need to read the index of the first neighbor ($n_0(j)$), then the index of the second neighbor ($n_1(j)$), and so on.

For sequential hardware (such as CPUs), it is then optimal to lay out the neighbors indices in memory grouping them by particle:

$$n_0(0), n_1(0), \dots, n_k(0), n_0(1), n_1(1), \dots, n_k(1), \dots \tag{12}$$

where $k+1$ is the maximum number of neighbors per particle. This will ensure that whenever a neighbor index is loaded from memory, the following neighbors indices will be cached too.

Additionally, better performance can be achieved by using a fixed-size neighbors list, obtained by precomputing the maximum number of neighbors that a particle can have: this allows the indices of the neighbors of each particle to be recoverable by simple computations, without additional memory access, at the expense of some wasted memory. A special marker can be added to the list of neighbors of each particle if it is not full, to indicate the actual end.

On GPUs, a more efficient memory layout is achieved when work-items $j, j+1, \dots$ load the index of their respective first neighbor from memory in adjacent memory locations. In this case it is therefore optimal to lay out the neighbors indices in memory grouping them by ordinal:

$$n_0(0), n_0(1), \dots, n_0(p), n_1(0), n_1(1), \dots, n_1(p), \dots \tag{13}$$

where p is the number of particles.

We call this layout *interleaved* or *neighbor-major*, in contrast to the *sequential* or *particle-major* layout used for CPUs, since from the perspective of each particle, the next neighbor is not found with an offset $+1$, but with an offset $+p$, with the memory in-between dedicated to the same neighbor ordinal for the other particles (Fig. 2).

The layouts we described work very well when the entire neighbors list must be traversed each time, but is sub-optimal when we want to only traverse a scattered subset of the list (for example, as we do in GPUSPH, considering only the neighbors of a given type).

To illustrate the approach we adopted to solve this issue, assume at first that we have only two particle types (fluid and boundary), and that when there are more neighbors of one type, there will be fewer of the other type. This is consistent with the fact that a particle far from the boundary will have a full neighborhood of fluid particles, but as it gets closer to the boundary the number of fluid neighbors will decrease, while the number of boundary neighbors will increase at a similar rate (assuming a more-or-less uniform particle distribution).

The solution for the split neighbors list in this case is to collect all neighbors of one type at the beginning of the list (from the first location up), and all the neighbors of the other type at the *end* of the list (from the last location down). If we denote by f_i the ith fluid neighbor and b_i the ith boundary neighbor, from the perspective of a single particle, the neighbors would then be stored as:

$$f_0, f_1, f_2, \dots, f_{k_f}, \odot, \dots, \odot, b_{k_b}, \dots, b_2, b_1, b_0 \tag{14}$$

where $k_f + 1$ is the number of fluid neighbors, $k_b + 1$ the number of boundary neighbors, and $\odot$ denotes the end-of-list marker. Of course this single-particle perspective can then be implemented system-wide using either the sequential layout, or with the interleaved layout.

The situation is more complex when the number of particle types grows. In the case of a third type, for example (as is the case for the semi-analytical boundary conditions with their vertex particles), the neighbors list will need to be split into two fixed-size chunks, one to store the first pair of types, and the other to store the remaining type (Fig. 3); indicating as before with f_i, b_i, v_i respectively the ith fluid, boundary and vertex neighbor, and with k_v the number of vertex neighbors, the list as seen by each particle would be coded as:

$$f_0, f_1, f_2, \dots, f_{k_f}, \odot, \dots, \odot, b_{k_b}, \dots, b_2, b_1, b_0, v_0, v_1, v_2, \dots v_{k_v} \odot \tag{15}$$

In this case, it is necessary to know the size of the chunk. In GPUSPH, we store the (local) index of the first boundary neighbors index, knowing that the vertex neighbors will start at the next location.

5. The compiler bottleneck

The use of more sophisticated C++ features in GPUSPH version 5 has driven us to test different compilers, with one of the primary motivations being the more developer-friendly error messages that modern C++ compilers provide. This has led us in particular to try Clang [68], that has been working on built-in support for CUDA for several years [69,70].

When the first Clang-compiled test case was run, a simple three-dimensional dam break, it was quite surprising to see a nearly 1.5× performance boost over the `nvcc`-compiled code . The performance gain was confirmed across multiple (recent) hardware generations and

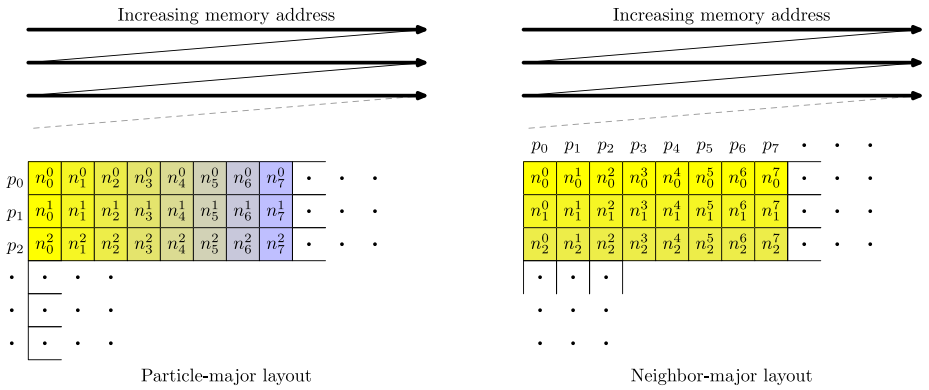

Fig. 2. Different layouts for the neighbors list, comparing the CPU-preferable "by particle" to the GPU-preferable "by neighbor" order.

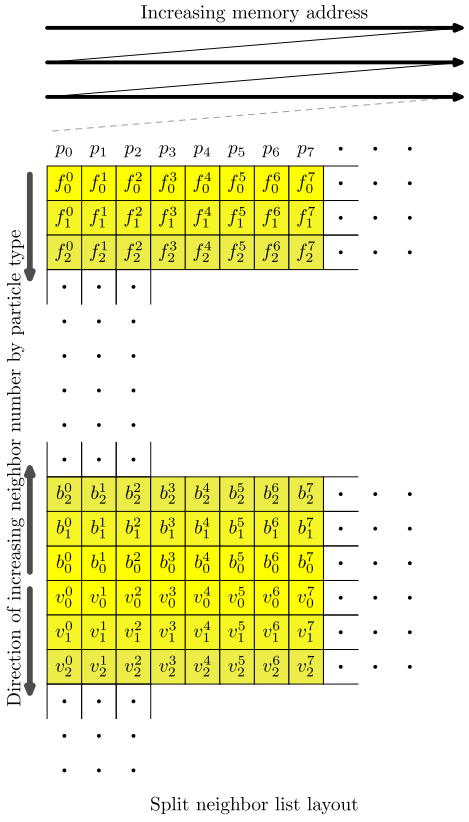

Fig. 3. Split-neighbors-list layout in the neighbor-major order case.

compiler versions, with tests spanning Clang version 9 to 12 and `nvcc` versions 10.1 to 11.4,

In what follows we will describe the analysis that led to the discovery of the source of this discrepancy, the changes we introduced to allow a similar performance gain across compilers, and the side-effect this had on multi-GPU support. Detailed benchmarks on the net effect of these improvements will be presented in Section 7.

5.1. Analysis of the performance difference

The test case used for the comparison is a three-dimensional dam-break with in a 1.6 m × 0.67 m × 0.6 m domain. The initial water volume occupies one side of the box, up to a height and depth of H = 0.4m. Framework options include Lennard-Jones boundary conditions for the solid walls, artificial viscosity, and the Molteni & Colagrossi [27] density diffusion model.

The simulations, consisting of 53,248 fluid and 30,388 boundary (for a total of 83,648) particles at a resolution of 32 particles in the H length (*ppH*), were only run for 1000 iterations, to gather basic information about total runtime and per-command contributions (Figs. 4–6). Data was not saved, since we were only interested in the computational performance. The results illustrated in the plots Figs. 4–6 refers to execution on an NVIDIA GeForce GTX 1650 Max-Q.

The first comparison was done between `nvcc` 11.4 and Clang 12. It was observed (Figs. 4 and 5), that many commands had in fact a marginal performance *regression* with Clang, with the most significant exception being the `FORCES_SYNC` command that runs the forces computation kernels, that take up the lion's share of the simulation (Fig. 6), thanks also to the fact that `FORCES_SYNC` and `EULER` are run twice per time-step, while the ancillary commands for neighbors list construction (`CALCHASH`, `SORT`, `REORDER`, `BUILDNEIBS`), are only run once every 10 time-steps.

A second surprising result came with the release of Clang 13, testing on which resulted in performances within less than 20% of those obtained when comping with `nvcc` (sometimes in excess, sometimes in defect, depending on test case and GPU architecture). Compared to the consistently improved performance of Clang 12 the results from Clang 13 were considered a regression in the compiler, and reported as an issue to the Clang developers [71].

A more thorough analysis of the forces kernels revealed that the key to the performance differences was in the usage of the stack. On GPU, use of the stack is particularly nefarious, as it involves the use of appropriately reserved global memory, which is no less than two orders of magnitude slower than registers, and can introduce significant latency in hot-path code. As mentioned before, reducing this kind of stack usage was indeed the reason for splitting the forces kernels into multiple functions, one for each particle type pair.

Two elements were thus surprising about the stack usage still being reported for the forces (and many other) kernels: the first was that, since all device functions are marked with the `always_inline` attribute in GPUSPH, there *should* have been no stack usage at all, once the excess variables had been eliminated; and the second was the question why Clang up to version 12 managed to avoid it, whereas the other compilers (all `nvcc` version and Clang 13 and higher) required it.

While the latter remains a mystery to date, resolved in Clang 14 by introducing an additional optimization pass in the compilation of CUDA code, the first question was answered by discovering the culprit in the virtual inheritance involved in the definition of the neighbors list iterators. Avoiding this was key to providing a significant performance boost across compilers.

5.2. The return of the neighbors list: multi-type iterators

The split neighbors list described in Section 4.1 makes it very efficient to traverse the list of a single neighbor type. An iterator simply

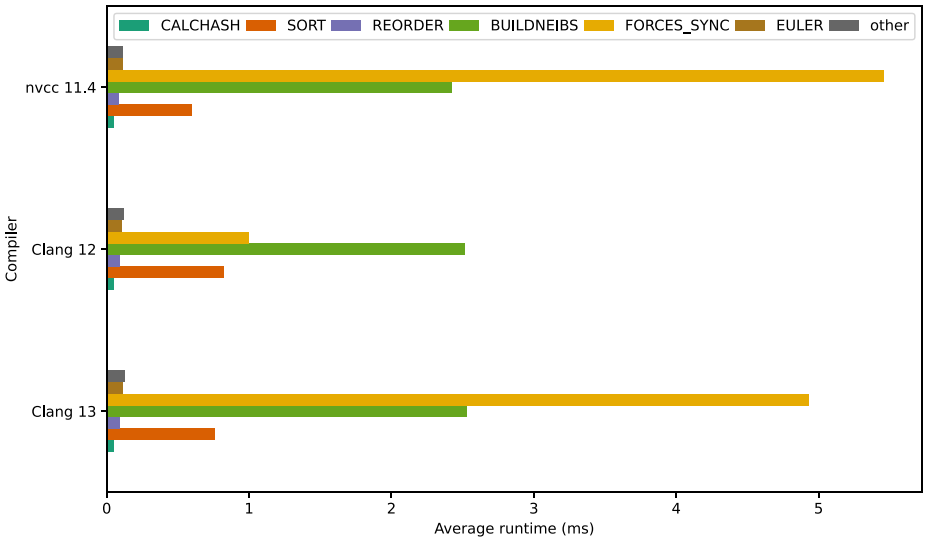

Fig. 4. Average runtime (in ms) per command invocation of the largest contributors, by compiler, with the initial code.

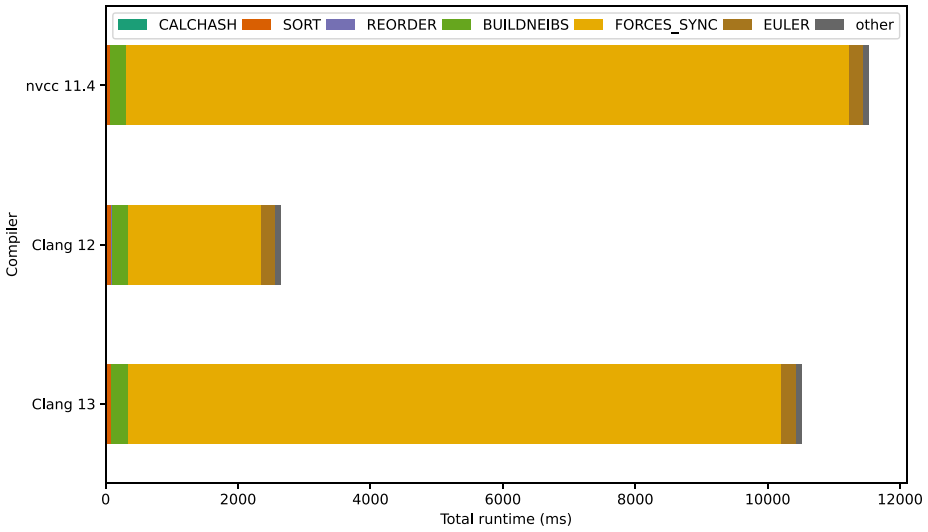

Fig. 5. Total cumulative runtime (in ms) for each command during the first 1000 simulation steps of the performance analysis test-case, by compiler, with the initial code.

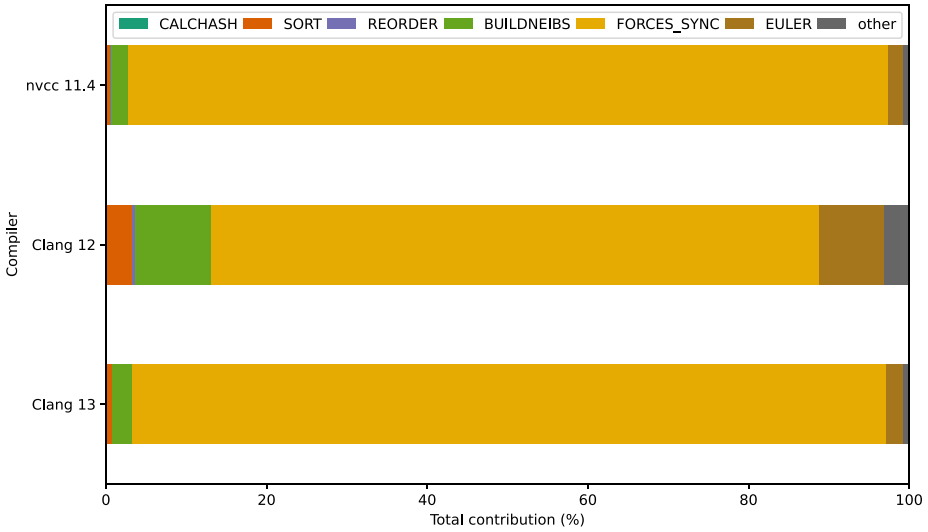

Fig. 6. Percent contribution by each command to the total runtime during the first 1000 simulation steps of the performance analysis test-case, by compiler, with the initial code.

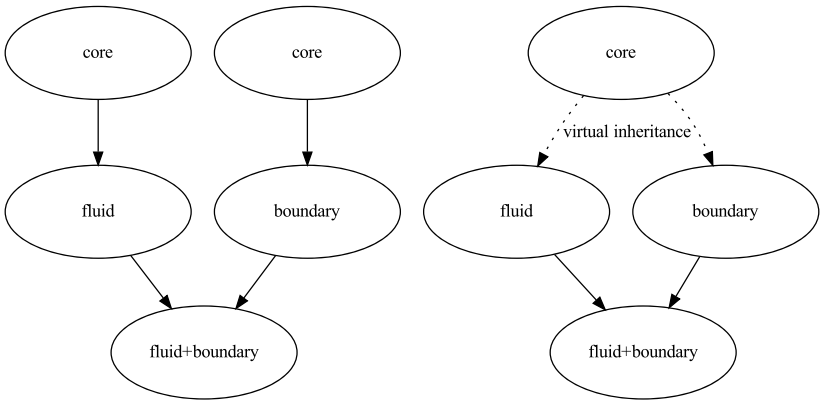

Fig. 7. Diamond inheritance of base classes with and without virtual inheritance through intermediate classes. In the standard case (left), two copies of the core class are inherited by the multi-type iterator, making any reference to `neibs_iterator_core` ambiguous. With virtual inheritance (right), a single copy of the core class is inherited, making it possible to access `neibs_iterator_core` unambiguous in the multi-type iterator.

needs to load the beginning of the chunk and the traversal direction (based on the neighbor type), and then load each neighbor index in sequence until the end-of-list marker is encountered.

When the same interaction needs to be computed with neighbors of different types (as is the case frequently with dynamic boundary conditions), the code necessary to traverse the neighbors list is made more complex by the need to change the offset and direction of traversal when one type is finished and the other begins, but aside from that the behavior remains essentially the same.

To abstract all this from the developer, we implemented `neibs_iterator` class templates that take care of all the internals, and present a simple interface with methods to retrieve the current neighbor's index, its relative position to the central particle, and finally a method to fetch the next neighbor and inform the caller when the neighbors list is completed.

The core of all the iterators is the same: a set of variables independent of the particle type, and the internal methods to fetch and decode the neighbor information from the neighbors list. These are abstracted in a dedicated `neibs_iterator_core` class.

All the single-type `neibs_iterator` class templates derive from the core class, and simply implement on top of it the necessary detail for the specific neighbor type, such as the computation of the chunk start offset and the traversal direction.

The straightforward way to implement a multi-type iterator, which is the solution we had adopted in version 5 of GPUSPH, is to create a dedicated class that derives from the corresponding single-type iterators, and switches from one to the other as the end-of-list marker is reached. This however leads to the infamous *diamond inheritance* problem: since all the single-type iterators depend on the (same) core class, and we want a single copy of the core class as (grand)parent of the multi-type iterator, the single-type iterators have to declare the core class as a *virtual* parent (Fig. 7).

This virtual inheritance, however, is in our case responsible for the inefficiency experienced with `nvcc` and Clang 13, as the compiler fails to fully de-virtualize the structure. Even worse, the negative impact of the virtual inheritance affects single-type iterators too, even though for them there would be no need for virtualization in the first place: indeed, single-type iterators inherit virtually from the core class only to support multi-type iterators. This is actually in conflict with one of the principles behind the GPUSPH design, that the implementation of a feature should not negatively affect other features (in this case, the possibility to iterate over multiple types should not affect the performance of iterating over a single type).

The new approach we have adopted in GPUSPH to avoid virtual inheritance and its negative effects on GPU performance has been to turn multiple inheritance into *chain* inheritance, leveraging the fact that we iterate over neighbor types in a predefined order (Fig. 8).

Instead of distinguishing between single- and multi-type iterators, we define a single class template for all iterators, with two template parameters: the particle type of the 'current' iterator, and the class of the 'next' iterator, with the template defining a class that derives from the 'next' iterator: this base class is then used to delegate the retrieval of the next neighbor when the current type is exhausted (Listings 1).

Listing 1: Nesting class template to implement multi-type neighbors list iterators.

```
template<ParticleType ptype, typename NextIterator>
class nested_neibs_iterator : public NextIterator
{
    using core = neibs_iterator_core;

    bool next() {
        if (core::current_type == ptype) {
            // fetch the next neighbor of this type
            bool has_next = fetch_next_neighbor();
            if (has_next) return true;
            // no more neighbors: switch to next type
            NextIterator::reset();
        }
        // this type has finished, delegate to next iterator
        return NextIterator::next();
    }

    /* rest of the class omitted for brevity */
};
```

By using the `neibs_iterator_core` as the "terminating" class, (i.e. the `NextIterator` when there are no more types to process), we ensure that all iterators inherit a single copy of it as (grand)parent through the chain, without having to resort to virtual inheritance. By adding to `neibs_iterator_core` a `reset()` method that invalidates `current_type` and a `next()` method that always returns false, the `nested_neibs_iterator` implements both single- and multi-type iterators with the same code, and the only distinction is given by the nesting of the classes (Fig. 8).

As show in Figs. 9–11, the effect of the de-virtualization of the neighbors iterator on the performances is impressive: the total runtime for `nvcc` and Clang 13 dropped from over 10 s to less than 5 s, bringing these compilers in line with the performance of Clang 12, that also benefits (although in much smaller amounts) from the optimization. Moreover, without the bottleneck created by the problematic virtual inheritance, the proprietary optimizations in `nvcc` allow the compiler to produce the fastest code among the ones we tested.

6. Too much of a good thing?

The impressive performance gains achieved by the rework of the neighbors traversal, that in some sense completes the split-neighbors implementation, have had some unintended consequences.

The first consequence has been an apparent decrease in the scaling efficiency of GPUSPH across multiple GPUs: this is due to the fact that

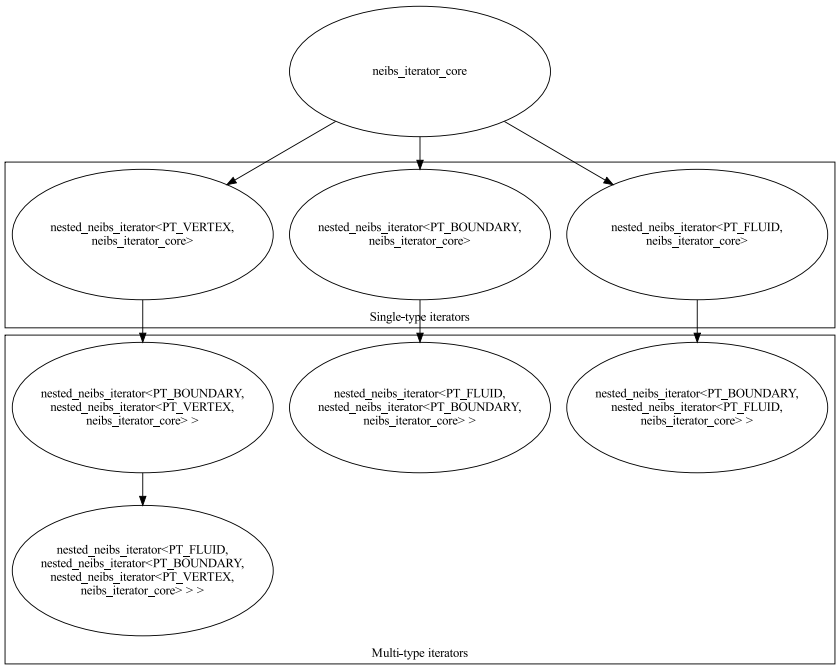

Fig. 8. Chain inheritance through nesting class templates. Single-type iterators derive directly from the core class, multi-type iterators derive from the other multi- or single-type iterators. For multi-type iterators, the order of the nesting determines the order in which neighbor types are processed. On the middle, for example, fluid types are processed before boundary types, whereas on the right boundary types are processed first.

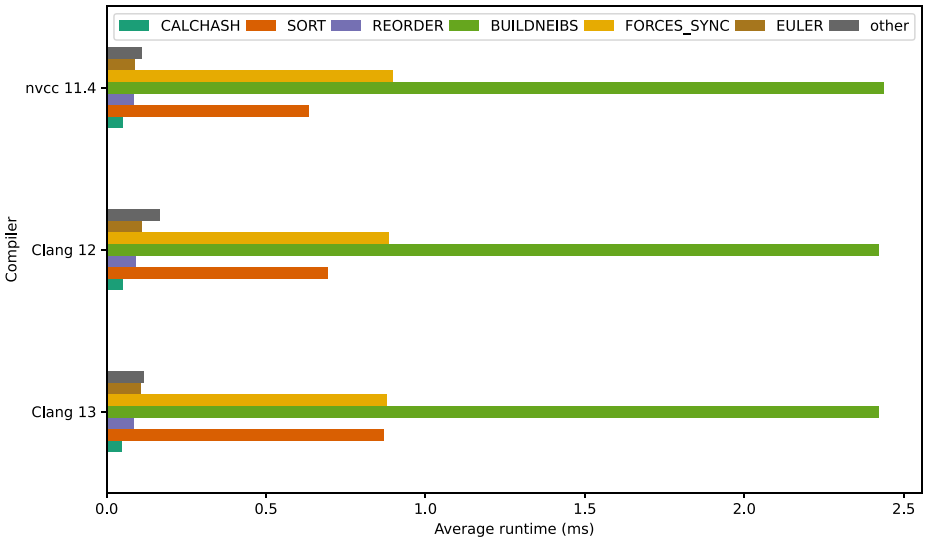

Fig. 9. Average runtime (in ms) per command invocation of the largest contributors, by compiler, with the devirtualized iterators.

high scalability relies on the forces computation taking enough time to cover the cross-device data transfer time (see [15,16] for details): with the forces computation being much faster, the amount of inner particles that need to be processed to cover data transfers has grown significantly, thus requiring much larger domains to achieve good scaling. This of course is compensated by the fact that a single GPU now manages to perform as well as 4 or 5 GPUs used to with the older code, thereby reducing the need for multi-GPU or multi-node to very large simulations, where the amount of inner particles becomes large enough to preserve the scaling properties. Further details with examples are discussed in Section 7.3

The second consequence of the forces kernels performance gain has more of a psychological than computational weight: as shown in Fig. 9, forces computation is not the slowest step of the simulation anymore: although forces computation is still the kernel where most time is spent overall (Fig. 11), its runtime for a single invocation is now comparable to the runtime of the neighbors list construction.

In fact, for smaller simulations such as the one shown in this analysis, a single run of the neighbors list construction can take more time than a single forces computation, with the discrepancy growing smaller as the number of particle increases, and forces computation still taking longer (although not much so) than neighbors list construction in the case of very large simulations.

In itself this effect is not particularly worrying, since the forces kernel runs twice per step (due to the predictor/corrector integration scheme), while the neighbors list construction only runs once every 10 steps (by default). As such, even when the two have comparable runtime, forces computation still weight 20 times more than the neighbors list construction, and optimization of the latter would be largely

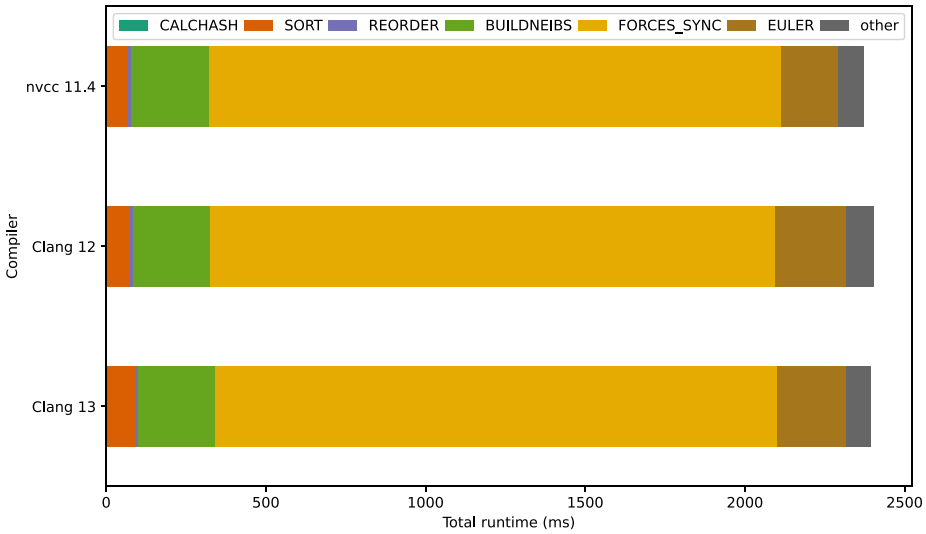

Fig. 10. Total cumulative runtime (in ms) for each command during the first 1000 simulation steps of the performance analysis test-case, by compiler, with the devirtualized iterators.

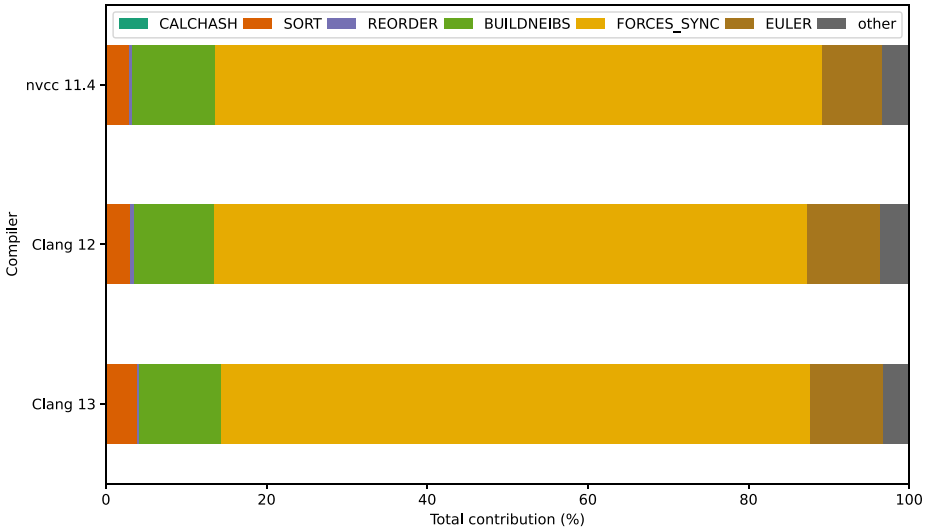

Fig. 11. Percent contribution by each command to the total runtime during the first 1000 simulation steps of the performance analysis test-case, by compiler, with the devirtualized iterators.

unnecessary, since even a 50% speed-up would only affect the total runtime of the simulation by less than 3%.

Still there is a psychological effect: knowing that building the neighbors list takes longer than a single forces computation is frustrating for a numerical code that *should* spend most of its time computing (numerical) derivatives. Hence the need to optimize the neighbors list construction.

6.1. Changing perspective: from the particle to the cell

As mentioned in Section 2.2, WCSPH with an explicit integrator lends itself to a natural parallelization where each work-item is associated with a particle. This is also true for the neighbors list construction: each particle looks for neighbors in the cells surrounding its own (Section 2.3), adding them to the neighbors list according to their type (Section 4.1).

This approach is trivial to implement, but leads to poor performance due to the scattered global memory accesses and consequent large latencies as the neighbors data is fetched, with the work-items unable to proceed until the data becomes available and decisions about the neighbors (Is it close enough? Which type does it belong to? Where does it go in the list?) must be made.

The first optimization is to take advantage of the split-neighbors layout to streamline the neighbors search. As particles are sorted in memory by cell, and within the cell by particle type (and finally by particle ID), we know that whenever looking at neighbors in a cell the central particle will first see all fluid particles, then all boundary particles, etc. Instead of a single loop going over all the particles in the cell, we can thus split the code into multiple loops, one per particle type.

Since the structure of the loop is largely the same, the loop can be implemented as a function template, parametrized on the neighbor type, taking care of all the neighbors of the given type, and returning when the next neighbor is of a different type (or there are no more neighbors). The instances of the function can be called in sequence, following the in-cell sort order (Listing 2). This change alone has been sufficient to give a performance gain in the order of 10% for the neighbors list construction, due to the reduction in divergence and to some control logic moving from runtime to compile time.

More significant improvements require a complete change in perspective. To explain how, we need to observe what happens at the

Listing 2: Traversing the neighbors in a cell by particle type.

```
template<ParticleType nptype, /* omissis */ >
void neibsInCellOfType
    (params_t const& params, var_t const& var, /* omissis */)
{
    /* from the neighbor loaded in var, keep processing all neighbors
     * until the next loaded one is not of the same type
     */
    for ( ; var.is_neib_of_type<nptype>(params); var.next_neib())
    {
        /* process this neighbor */
    }
}

template<typename params_t, /* omissis */>
void neibsInCell(params_t const& params, /* omissis */)
{
    /* omissis */

    var_t vars(params, /* omissis */);

    // Load information about the first neighbor
    var.load_neib_info(params);

    // Process neighbors by type, leveraging the fact that
    // within cells they are sorted by type.
    // Each instance of neibsInCellOfType will process the neighbors
    // of the given type, and return when the next loaded neighbor
    // (if any) is of a different type
    if (var.neib_type == PT_FLUID)
        neibsInCellOfType<PT_FLUID, /* omissis */>
            (params, var, /* omissis */);

    if (var.neib_type == PT_BOUNDARY)
        neibsInCellOfType<PT_BOUNDARY, /* omissis */>
            (params, var, /* omissis */);

    if (boundarytype == SA_BOUNDARY && var.neib_type == PT_VERTEX)
        neibsInCellOfType<PT_VERTEX, /* omissis */>
            (params, var, /* omissis */);

};
```

hardware level during the execution of the standard implementation on GPU.

Consider two particles with consecutive indices. In many cases, they will belong to the same cell, and process neighboring cells concurrently. In fact, with the per-particle perspective adopted so far, they will process the same particles in the neighboring cells concurrently: they will load at the same time the first particle in the cell at offset $(-1,-1,-1)$, decide what to do with it, then move (at the same time) to the second particle in the same cell, etc, until the neighboring cell is exhausted, and then move to the next cell.

This is not an ideal access pattern: even assuming that the memory controller supports broadcasting (so the neighboring particle data is fetched for all work-items at the same time), the effective bandwidth is limited, since each transaction only transfers the data of a single particle, whereas the controller on GPUs is designed to transfer data from *consecutive* memory locations to *consecutive-index* work-items (which is the reason why the neighbors list is stored in interleaved format on GPU, as explained in Section 4.1).

To maximize throughput, we would need work-items to transfer data corresponding to *adjacent* particles: for example, while the work-item for particle i requests the data for the first particle from the cell at offset $(-1,-1,-1)$, the work-item from particle $i+1$ should request the data for the *second* particle in the cell, and so on. Of course, all work-items would then need the information from the neighboring cell particles, which have been loaded by *other* work-items: this data can be exchanged by storing it on fast on-chip shared memory (accessible by all the work-items in the same work-group).

To maximize the usefulness of this approach, we can switch from a particle-based perspective to a cell-based perspective, leveraging the fact that on NVIDIA GPUs work-items (a.k.a. *CUDA threads*) do not execute independently, but proceed in lock-step grouped in 32-wide sub-groups (a.k.a. *warps*).

The idea is thus to map each warp to a cell, and each work-item in the warp to a particle in the cell. This *guarantees* that all work-items in the same warp traverse the same neighboring cells, in lock-step. Each work-item (from the same warp) then loads one particle from the neighboring cell into the shared memory, and all work-items (in the warp) process the neighbors taking the data from the shared memory.

Compared to the naive thread-to-particle mapping, the warp-to-cell mapping is considerably more complex to implement. One of the main difficulties is that it is not possible to do early bail-outs: even work-items that are not associated to active particles must process the neighboring cells, to ensure that all neighbors data is correctly loaded; this requires additional boolean variables to be carried around, controlling whether the work-item must also process the neighbors, or only contributes to the data retrieval.

Additionally, a cell may contain more particles than there are work-items in the warp. Therefore, both the central particle selection and the neighbor data loads are run in a "sliding window" fashion, until exhaustion of the cell particles.

That being said, in the most common configuration of SPH smoothing kernels with radius $k = 2$ and smoothing factors of $h = 1.3$, cells in three-dimensional problems have less than 32 particles. This actually leads to a sub-optimal usage of the hardware, since every running warp will have some work-items masked out (typically, only the first 20 to 24 work-items in each warp will be active), leading to a hardware efficiency of 75% *or less*, compared to the particle-based approach where all work-items will be active and running.

Despite this inefficiency, however, the cell mapping comes out to be around 50% *faster* than the particle-based approach, due to the significant improvements in memory access patterns and the consequent reduction in latency. This is consistent with Volkov's findings about the relative importance of latency and occupancy in gauging parallel algorithms and their performance on GPU [72,73].

This performance gain (computed as the relative change in runtime, i.e. $T_{\text{old}}/T_{\text{new}} - 1$, where $T_{\text{old}}, T_{\text{new}}$ are the runtimes of the old and new code) in the neighbors list construction runtime only has a modest effect on the total runtime (Fig. 12), diminishing as the number of particles grows larger.

As mentioned, this is expected, due to the lower frequency of execution of the neighbors list construction in typical simulations, but the benefits can become important if the list needs to be updated more frequently.

7. An industrial test case

The performance gains presented so far are crucial in industrial applications of SPH, for which hundreds of millions of particles are typically employed, due to large domain sizes and/or fine resolutions.

We illustrate this in a real-world test case, showing performance results including single-node and multi-node scaling performance before and after the recent optimizations, and validation of the simulation results against experimental data.

7.1. Test case setup

The test case refers to a large-scale wave maker and basin with a bounding box that is approximately 65 m × 85 m (Fig. 13). The wave is generated by a caisson system to produce a nominal maximum wave height $H = 0.9$ m with period $T = 7.5$ s. The water depth at the caisson is $d = 3.4$ m and decreases to 0 m at the end of the basin with a non-uniform reef bathymetry in-between.

For the SPH simulations, the Lennard-Jones boundary model was used, with a geometric description of the basin bathymetry through a Digital Elevation Model exerting a point-wise normal Lennard-Jones force [7]. The water is assumed inviscid, and no turbulent or artificial viscosity terms are included in the momentum equation. The Molteni & Colagross density diffusion model [27] is used. The chosen SPH kernel is the Wendland quintic with radius 2, and smoothing factor 1.3.

The hardware used for the run consists of 3 nodes with 3 RTX 3090 GPUs each. Each GPU is equipped with 24 GB of GDDR6X VRAM running at 1.2 GHz wit a peak theoretical bandwidth of 936 GB/s,

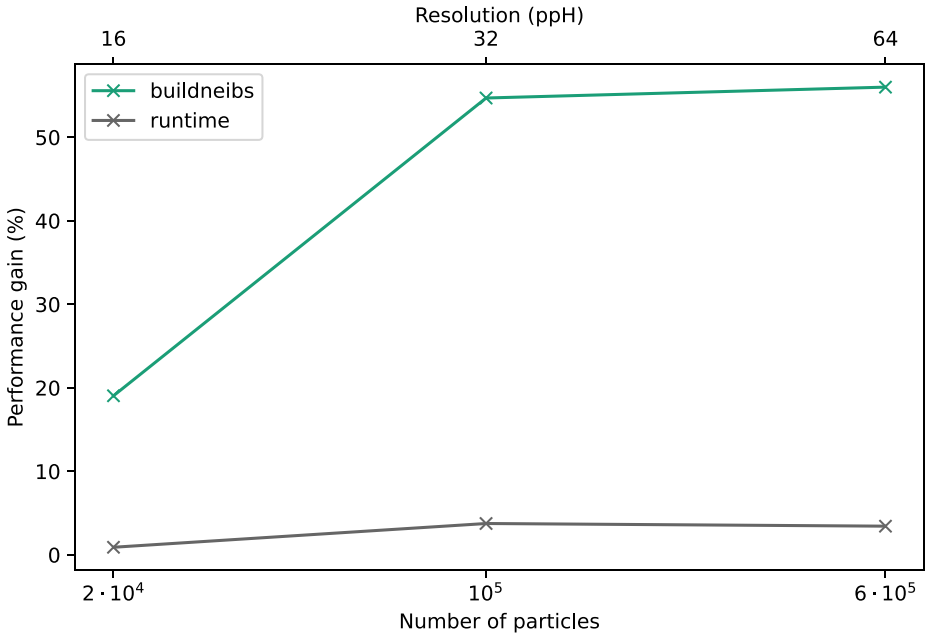

Fig. 12. Performance gain (see the text for the details) in the average runtime for the neighbors list construction and in the total runtime, using the per-cell approach over the per-particle approach, with the `nvcc` compiler.

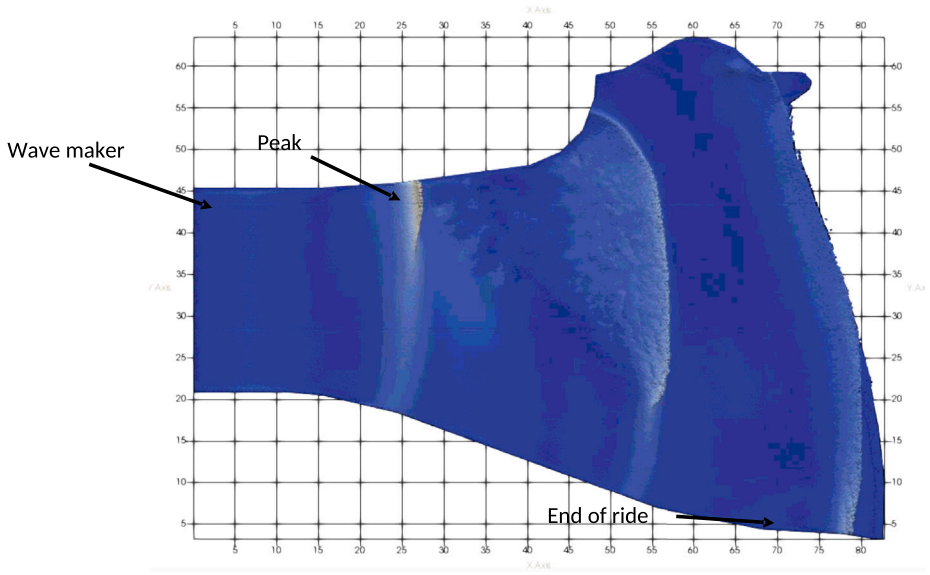

Fig. 13. Overhead view of the irregularly-shaped wave basin with variable bottom bathymetry used in the test case, including the location of the three wave gages used for the comparison between the experimental and numerical data.

and 82 compute units with 128 shader processors each for a total of 10,496 "CUDA cores" per GPU, with a peak theoretical throughput of around 30 TFLOPS for single-precision FMA. Due to the hardware setup, peering between GPUs within a node was unavailable, lowering the multi-GPU scaling performance, as discussed below. The nodes are equipped with 1 Gbps network cards and are interconnected through a 24-port switch.

7.2. Validation

Comparison between the physical experiments and SPH simulations were done comparing the water surface elevation and time-of-arrival at three distinct gage positions: near the caisson, where the peak wave height is expected, and at the end of the ride (Fig. 14). Simulations were run at three different resolutions, with inter-particle spacing respectively $\Delta p = 0.01, 0.008, 0.005$m resulting respectively in (fluid + boundary = total) $4 + 2.4 = 6.4, 7.7 + 3.8 = 11.5$ and $32 + 10 = 42$ million particles.

Even though the chosen options result in a quite simple model, the results show good accuracy. Thanks to the choice of density diffusion model, the simulation is stable over the simulated period despite the absence of viscous dissipation in the momentum equation. As expected, the match of the wave timing and crest height improve at higher resolutions. The largest discrepancy is seen at the end-of-ride gage,

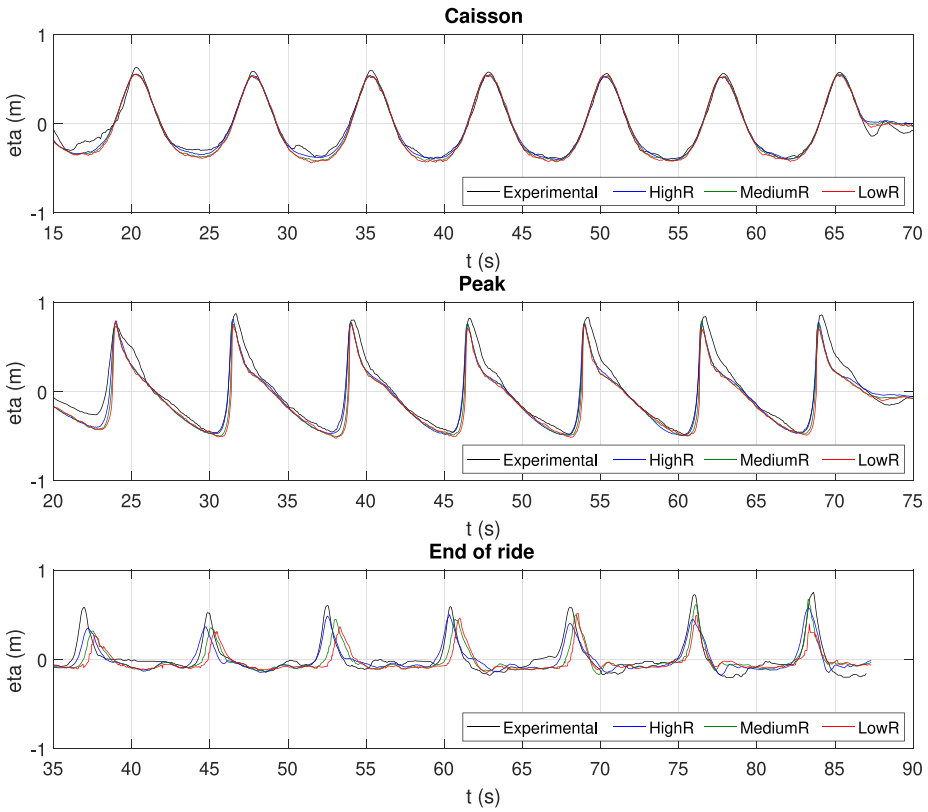

Fig. 14. Water surface elevation over time from the experimental data compared with the low, medium and high resolution SPH simulations at the caisson (top figure), peak (middle figure) and end-of-ride (bottom figure).

where excessive dissipation is observed even at higher resolutions. This effect may be reduced with the adoption of better conservation models such as the CCSPH model presented in [32], at the cost of higher computational requirements.

7.3. Performance results

Metrics. Performance in GPUSPH is measured as of iterations times particles per second (IPPS), or more commonly in MIPPS (10^6 IPPS), a metric we first introduced in [16]. This can be easily computed for any implementation as the number of particles in the simulation, multiplied by the number of iterations, and divided by the total runtime. (GPUSPH provides real-time feedback on the performance during execution.)

A rough conversion from IPPS to the PIPS (particle interactions per second) metric used by other software [74] can be achieved, at least for simpler boundary models, multiplying the value in MIPPS by the average number of neighbors per particle, and then by 2 to account for the predictor/corrector integration scheme. For example, 100 MIPPS on a simulation with an average of 80 neighbors per particle corresponds approximately to 16 GPIPS (1 GPIPS being 10^9 PIPS).

Single-GPU. In our performance analysis we start by comparing the single-GPU performance across different resolutions before and after introducing the neighbors list construction and traversal optimizations. As show in Fig. 15, the performance ratio for this test case at the tested resolutions is essentially constant, with the optimizations bringing a consistent 4× speed-up to the code. The slight decrease in MIPPS at higher resolutions is due to the higher percentage of fluid particles in the total particle count, and the higher computational requirements of fluid particles, for which the physical equations of motions have to be solved, over boundary particles, that only contribute to the forces applied to the fluid particles in their influence sphere.

Multi-GPU. We can evaluate the strong scaling capability of GPUSPH by comparing the performance of our code at given resolutions on a growing number of device. This is usually achieved by looking at the ratio $T_1/(nT_n)$ where T_i represents the runtime with i devices. As our preferred unit of measure is the number of iterations times particles per second, the scaling efficiency can be likewise measured as $P_n/(nP_1)$ where P_i is the number of MIPPS achieved with i devices.

Due to the embarrassingly parallel nature of WCSPH with explicit integration – as used in our tests – the only major factor impacting scalability comes from the latency introduced between solver steps when transferring data about neighbors to/from other devices. As explained in [15,16], each device in GPUSPH holds a copy of the data belonging to neighboring particles ("halo" particles) that reside on other device, and this is updated after each computational kernel, with the exception of the integration, which is a simple increment without any neighbors list traversal, and is therefore done on the halo particles with the forces data transferred from the neighboring devices after the forces computation kernel.

Historically, the forces computation kernel has been the most computationally expensive kernel, and this has been used to minimize the impact of data transfer on scaling performance, by first computing the forces on the particles to be transferred to other devices, and then running the computation on the rest of the domain (inner particles) while the computed forces are being copied. This allowed GPUSPH to achieve nearly ideal scaling [16,75] under the condition that the time needed to transfer the halo particles data is less than the time needed to compute the forces on the inner particles.

This typically requires fast device-to-device transfers (e.g. through peering for devices on the same node, or solutions such as GPU Direct in multi-node configurations), and a sufficient number of inner particles. With the speed-up of the forces kernel computations coming from the optimized neighbors traversal code, we can expect the scaling performance of the new code to degrade compared to the scaling

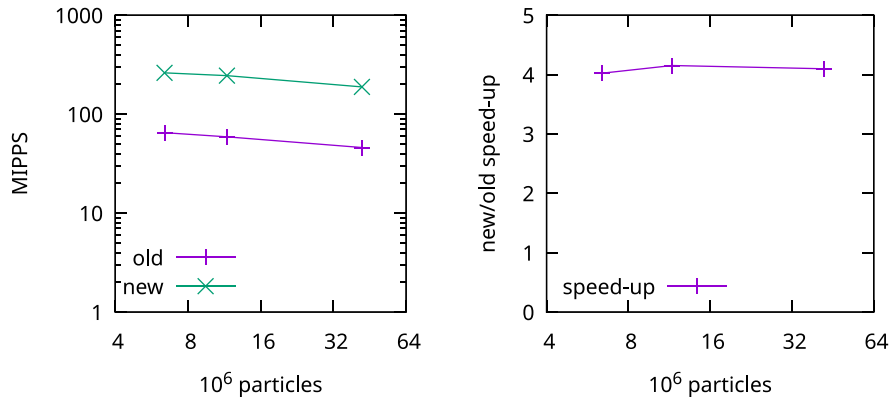

Fig. 15. Single-GPU performance with the old (unoptimized) and new (optimized) neighbors list construction and traversal code. Left: performance (in MIPPS). Right: performance ratio.

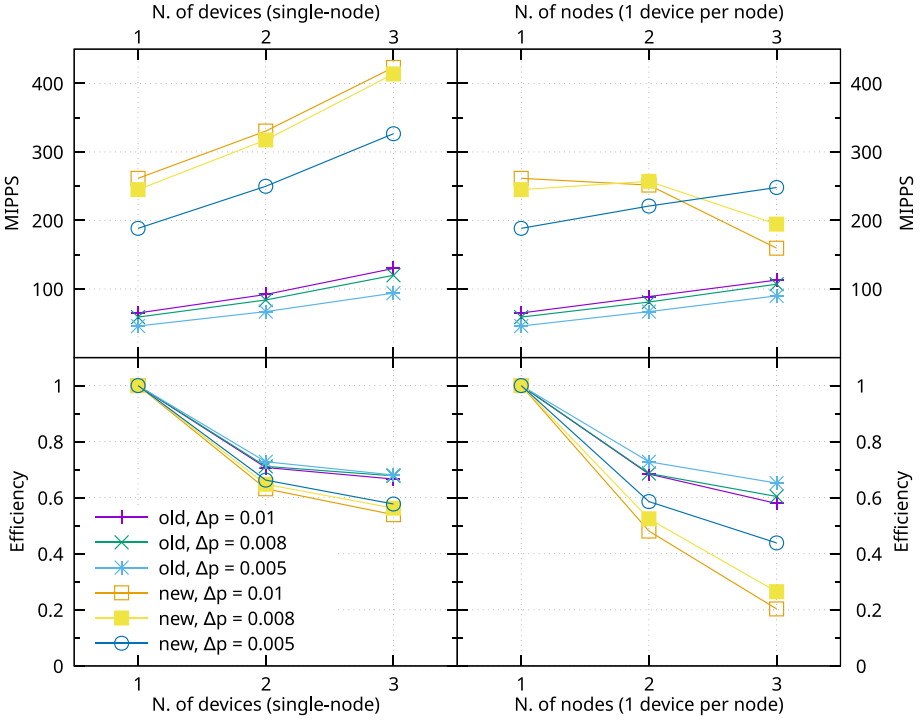

Fig. 16. Performance (top) and strong scaling (bottom) at different resolutions. The left plots refer to the multi-GPU single-node case. The right plots refer to the multi-node case, with 1 GPU per node.

performance of the old code, especially when the number of particles (and particularly inner particles) is small. The effect will be particularly evident in our tests due to the hardware configuration that prevents device peering, and even more so in the multi-node cases due to the higher latency and lower bandwidth of network data transfers compared to transfers within the same node.

The results for 1 to 3 devices are shown in Fig. 16, separating the single-node multi-GPU results (with all devices attached to the same host machine) from the multi-node results, limited in this case to 1 GPU per node.

In the single-node case, where data transfers are more efficient, all tests show performance growing with the number of GPUs. The old code has lower performance, but better scaling, with an efficiency of around 70%, whereas the new code drops to an efficiency as low as 53% in the low resolution case (57% at the highest resolution).

This is due to the much faster forces computation kernel in the new version of the code failing to fully cover the latency introduced by the data transfer. The effect is even more evident in the multi-node, 1 GPU per node case, where the overall performance of the code actually *drops* as the number of nodes grows, with the only exception being the highest resolution case, where the workload for the forces computation kernel is sufficient to cover at least part of the network data transfers. Both versions of the code suffer from the more expensive data transfers, with the old code efficiency dropping to between 58% (low resolution) and 65% (high resolution), and the new code only managing 44% efficiency at high resolution, with the low resolution dropping to 20%.

Due to the insufficient computational load in the lower resolution cases, we only test the configuration with up to 9 devices (3 nodes with 3 devices each) in the high resolution case. The results are shown in Fig. 17. Increasing the number of devices in this case always results in higher performance (and thus faster simulations), but there are diminishing returns as the number of particles assigned to each device decreases.

Once again the effect is more evident with the new, faster code, that needs a larger number of particles for the forces kernel computation to take enough time to cover the data transfers, especially in the multi-node, 3 GPUs per node configurations. Given the scaling of the forces

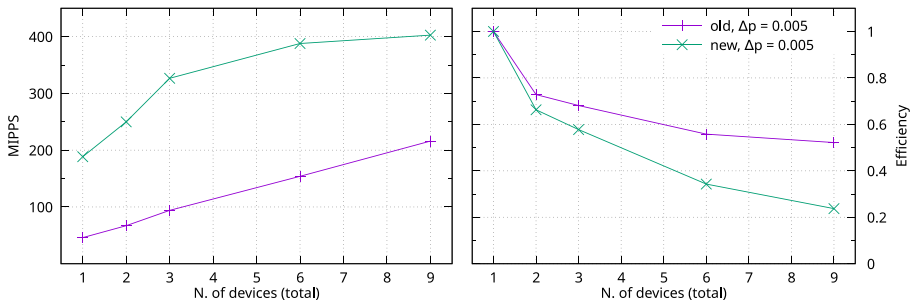

Fig. 17. Performance (left) and strong scaling (right) for the high resolution case. A single node is used for up to 3 devices, with 6 and 9 devices using 2 and 3 nodes respectively.

kernel, we would expect no less than 6× the number of inner particles per GPU to be needed to achieve scaling efficiency comparable to that of the old code.

8. Discussion and conclusions

Neighbor list construction and traversal are critical paths in SPH code and essential for optimal performance. This issue is compounded in software that defines multiple particle types, where a high-level interface is necessary to help developers, and the data structures must still be optimized for every use-case.

On GPUs, where some programming patterns may result in excessive use of the main memory rather than hardware registers to hold temporary data, catching or missing implementation-specific optimization opportunities can change the performance of individual kernels by nearly an order of magnitude. As a result, compiler choice can be at least as important as the coding strategies in delivering the best performance on any given hardware, although significant differences between compilers are a likely indication of "code smells", i.e. sections of the code that are still using sub-optimal, and thus harder to optimize, patterns.

We have observed this when introducing support for different toolchains in GPUSPH, which has highlighted a previously unknown `nvcc`-related bottleneck in the neighbors traversal code. This has led us to rethink the types used to represent the neighbors list iterators, resulting in performance gains measurable in a factor of 3 ± 1 on GPU, depending on problem size, with higher speed-ups achieved in test cases with more particles.

A paradox with increasing performance is a potential drop in the strong scaling efficiency in multi-GPU (especially multi-node) simulations, as the shorter execution time for forces computation can fail to fully cover the time needed to transfer data between devices. This is more than compensated by the shorter computational times in general, and may imply that a much higher per-device load is needed to achieve comparable efficiency. Hardware setups with faster connection speeds and lower latency can also help improve scaling.

The reduced strong scaling efficiency we observed can also be viewed from a different perspective, as it is the byproduct of better hardware utilization in the single-GPU case: what could saturate the compute units with the old code is not sufficient anymore. Beyond the apparently negative surface, we still see how beneficial this is with an example. Consider the case of a simulation that scales with 75% efficiency on 4 GPUs with the old code, and with only 50% efficiency with the new code, but with a 4× overall speedup. If the simulation takes 24 h on a single GPU with the old code, distributing across the 4 GPUs would reduce the runtime to $24/4/0.75 = 8$ h, whereas running it on a single GPU with the new code would reduce the runtime to 6 h. A single-GPU run with the new code is already faster than a 4 GPUs run with the old code. Thanks to the much lower baseline, even with 50% scaling on 4 GPUs, this could still be lowered to 3 h. And that is without considering that scaling improves for larger simulations.

The improvements to the neighbors traversal have also highlighted that, as the forces kernel computation efficiency improves, the cost of the neighbors list construction itself (or in its absence the neighbors search implementation) increases, even if weighted by the reduced frequency at which it is run. We have shown here that, as an intrinsically memory-bound procedure, neighbors search and list construction can benefit from a change in perspective that leads to improved coalescence of memory transactions and better caching on GPU. In our experience, switching from the naive per-particle vision to an aggregate per-cell approach can lead to performance gains of as much as 50% in the neighbors search despite a lower hardware occupancy.

Similar optimizations, which we will explore in the future, could also be possible for the main computational kernels implementing the actual numerical method, indicating that despite the appeal of SPH as an embarrassingly parallel method, optimal performance lies behind non-trivial implementation strategies.

The programming strategies we have presented are specific to the implementation details of GPUSPH. but we believe that our experience is indicative of a more general existence of untapped potential to improve scientific codes that goes beyond just hardware acceleration and novel algorithms. The fast evolution of GPU architectures is often a drive to revisit implementations for further optimizations, but such opportunities may be available on CPU as well, especially considering the ongoing progress in compiler technology and programming languages, and opportunities such as the ones illustrated in this paper may be worth exploring, both in research and for engineering applications, given the potential for significant speed-ups.

CRediT authorship contribution statement

Giuseppe Bilotta: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Conceptualization. **Vito Zago:** Writing – review & editing, Validation, Software. **Alexis Hérault:** Writing – review & editing, Software. **Annalisa Cappello:** Writing – review & editing, Software. **Gaetana Ganci:** Writing – review & editing, Software. **Hendrik D. van Ettinger:** Writing – review & editing, Validation. **Robert A. Dalrymple:** Writing – review & editing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Lucy LB. A numerical approach to the testing of the fission hypothesis. Astron J 1977;82:1013–24.

[2] Gingold RA, Monaghan JJ. Smoothed particle hydrodynamics - theory and application to non-spherical stars. Mon Not R Astron Soc 1977;181:375–89.

[3] Monaghan JJ. Smoothed Particle Hydrodynamics. Rep Progr Phys 2005;68:1703–59. http://dx.doi.org/10.1088/0034-4885/68/8/R01.

[4] Zago V, Bilotta G, Cappello A, Dalrymple RA, Fortuna L, Ganci G, Hérault A, Del Negro C. Simulating complex fluids with smoothed particle hydrodynamics. Ann Geophys 2017;60:PH669. http://dx.doi.org/10.4401/ag-7362.

[5] Cleary PW, Monaghan JJ. Conduction modelling using smoothed particle hydrodynamics. J Comput Phys 1999;148:227–64. http://dx.doi.org/10.1006/jcph.1998.6118.

[6] Monaghan JJ, Huppert HE, Worster MG. Solidification using smoothed particle hydrodynamics. J Comput Phys 2005;206:684–705. http://dx.doi.org/10.1016/j.jcp.2004.11.039.

[7] Bilotta G, Hérault A, Cappello A, Ganci G, Del Negro C. GPUSPH: a Smoothed particle hydrodynamics model for the thermal and rheological evolution of lava flows. In: Detecting, modelling and responding to effusive eruptions. geological society of London. Geological society, volume 426, London: Special Publications; 2016, p. 387–408. http://dx.doi.org/10.1144/SP426.24.

[8] Pharr M, Fernando R, editors. GPU gems 2: programming techniques for high-performance graphics and general-purpose computation. Addison-Wesley; 2004.

[9] Harada T, Koshizuka S, Kawaguchi Y. Smoothed particle hydrodynamics on GPUs. Comput Graph Int 2007;6:3–70.

[10] Zeller C, editor. NVIDIA CUDA programming guide, version 1.0. NVIDIA Corporation; 2007.

[11] Hérault A, Bilotta G, Dalrymple RA. SPH on GPU with CUDA. J Hydraul Res 2010;48:74–9.

[12] Cercos-Pita J. AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL. Comput Phys Comm 2015;192:295–312. http://dx.doi.org/10.1016/j.cpc.2015.01.026, URL: https://www.sciencedirect.com/science/article/pii/S0010465515000909.

[13] Crespo A, Domínguez J, Rogers B, Gómez-Gesteira M, Longshaw S, Canelas R, Vacondio R, Barreiro A, García-Feal O. DualSPHysics: Open-source parallel CFD solver based on smoothed particle hydrodynamics (SPH). Comput Phys Comm 2015;187:204–16. http://dx.doi.org/10.1016/j.cpc.2014.10.004, URL: https://www.sciencedirect.com/science/article/pii/S0010465514003397.

[14] Zhang C, Rezavand M, Zhu Y, Yu Y, Wu D, Zhang W, Wang J, Hu X. SPHinXsys: An open-source multi-physics and multi-resolution library based on smoothed particle hydrodynamics. Comput Phys Comm 2021;267:108066. http://dx.doi.org/10.1016/j.cpc.2021.108066, URL: https://www.sciencedirect.com/science/article/pii/S0010465521001788.

[15] Rustico E, Bilotta G, Hérault A, Del Negro C, Gallo G. Advances in multi-GPU smoothed particle hydrodynamics simulations. IEEE Trans Parallel Distrib Syst 2014a;25:43–52, http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.340.

[16] Rustico E, Jankowski JA, Hérault A, Bilotta G, Del Negro C. Multi-GPU, multi-node SPH implementation with arbitrary domain decomposition. In: Proceedings of the 9th SPHERIC workshop. Paris; 2014b., p. 127–33.

[17] Domínguez JM, Crespo AJC, Gómez-Gesteira M, Marongiu JC. Neighbour lists in smoothed particle hydrodynamics. Internat J Numer Methods Fluids 2011;67:2026–42. http://dx.doi.org/10.1002/fld.2481.

[18] Bilotta G, Zago V, Hérault A. Design and implementation of particle systems for meshfree methods with high performance. In: Chickerur S, editor. High performance parallel computing. Rijeka: IntechOpen; 2018, http://dx.doi.org/10.5772/intechopen.81755, chapter 6.

[19] Bilotta G, Zago V, Hérault A, Saikali E, Dalrymple RA. Bigger, cleaner, faster: notes on the implementation of a powerful, flexible, high-performance SPH computational engine. In: Proceedings of the 14th SPHERIC workshop. UK: Exeter; 2019.

[20] Stroustrup B. Programming: principles and practice using C++. second ed.. Addison–Wesley; 2014.

[21] Cole RH. Underwater explosion. Princeton, NJ: Princeton University Press; 1948.

[22] Batchelor GK. An introduction to fluid mechanics. Cambridge University Press; 1974.

[23] Grenier N, Antuono M, Colagrossi A, Le Touzé B. An Hamiltonian interface SPH formulation for multi-fluid and free surface flows. J Comput Phys 2009;228:8380–93. http://dx.doi.org/10.1016/j.jcp.2009.08.009.

[24] Hu XY, Adams NA. A multi-phase SPH method for macroscopic and mesoscopic flows. J Comput Phys 2006;213:844–61. http://dx.doi.org/10.1016/j.jcp.2005.09.001.

[25] Colagrossi A, Landrini M. Numerical simulation of interfacial flows by smoothed particle hydrodynamics. J Comput Phys 2003;191:448–75. http://dx.doi.org/10.1016/S0021-9991(03)00324-3.

[26] Morris JP, Fox PJ, Zhu Y. Modeling low reynolds number incompressible flows using SPH. J Comput Phys 1997;136:214–26. http://dx.doi.org/10.1006/jcph.1997.5776.

[27] Molteni D, Colagrossi A. A simple procedure to improve the pressure evaluation in hydrodynamic context using the SPH. Comput Phys Comm 2009. http://dx.doi.org/10.1016/j.cpc.2008.12.004.

[28] Marrone S, Antuono M, Colagrossi A, Colicchio G, Le Touzé G. δ-SPH model for simulating violent impact flows. Comput Methods Appl Mech Engrg 2011;200:1526–42. http://dx.doi.org/10.1016/j.cma.2010.12.016.

[29] J.K. Chen, Beraun JE, Jih CJ. An improvement for tensile instability in Smoothed Particle Hydrodynamics. Comput Mech 1999;23:279–87. http://dx.doi.org/10.1007/s004660050409.

[30] Vila JP. On particle weighted methods and smooth particle hydrodynamics. Math Models Methods Appl Sci 1999;9. http://dx.doi.org/10.1142/S0218202599000117.

[31] Guilcher PM, Ducrozet G, Alessandrini B, Ferrand P. Water wave propagation using SPH models. In: 2nd SPHERIC workshop, 2007. Madrid, Spain; 2007.

[32] Zago V, Schulze LJ, Bilotta GA, Dalrymple RA. Overcoming excessive numerical dissipation in SPH modeling of water waves. Coast Eng 2021;170:104018. http://dx.doi.org/10.1016/j.coastaleng.2021.104018, URL: https://www.sciencedirect.com/science/article/pii/S0378383921001666.

[33] Vacondio R, Altomare C, Leffe MDe, Hu X, Le Touzé D, Lind S, Marongiu JC, Marrone S, Rogers BD, Souto-Iglesias A. Grand challenges for Smoothed particle hydrodynamics numerical schemes. Comput Part Mech 2021;8:575–88. http://dx.doi.org/10.1007/s40571-020-00354-1.

[34] Liu GR, Liu MB. Smoothed particle hydrodynamics: a meshfree particle method. World Scientific Publishing Company; 2003.

[35] Monaghan JJ, Kajtar JB. SPH particle boundary forces for arbitrary boundaries. Comput Phys Comm 2009;180:1811–20. http://dx.doi.org/10.1016/j.cpc.2009.05.008.

[36] Dalrymple RA, Knio O. SPH modelling of water waves. In: Coastal dynamics 2001. 2001, p. 779–87.

[37] Crespo AJC, Gómez-Gesteira M, Dalrymple RA. Boundary conditions generated by dynamic particles in SPH methods. Comput Mater Continua 2007;5:173–84.

[38] Adami S, Hu XY, Adams NA. A generalized wall boundary condition for Smoothed Particle Hydrodynamics. J Comput Phys 2012;231:7057–75. http://dx.doi.org/10.1016/j.jcp.2012.05.005.

[39] Ferrand M, Laurence DR, Rogers BD, Violeau D, Kassiotis C. Unified semi-analytical wall boundary conditions for inviscid, laminar or turbulent flows in the meshless SPH method. Internat J Numer Methods Fluids 2012;71:446–72. http://dx.doi.org/10.1002/fld.3666, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.3666, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.3666.

[40] Mayrhofer A, Ferrand M, Kassiotis C, Violeau D, Morel FX. Unified semi-analytical wall boundary conditions in SPH: analytical extension to 3-D. Numer Algorithms 2015;68:15–34. http://dx.doi.org/10.1007/s11075-014-9835-y.

[41] Kostorz W, Esmail-Yakas A. Semi-analytical smoothed-particle hydrodynamics correction factors for polynomial kernels and piecewise-planar boundaries. Internat J Numer Methods Engrg 2021;122:7271–305. http://dx.doi.org/10.1002/nme.6771, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.6771, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.6771.

[42] Vela Vela L, Reynolds-Barredo J, Sánchez R. A positioning algorithm for sph ghost particles in smoothly curved geometries. J Comput Appl Math 2019;353:140–53. http://dx.doi.org/10.1016/j.cam.2018.12.021, URL: https://www.sciencedirect.com/science/article/pii/S0377042718307520.

[43] Ferrand M, Joly A, Kassiotis C, Violeau D, Leroy A, Morel FX, Rogers BD. Unsteady open boundaries for SPH using semi-analytical conditions and Riemann solver in 2D. Comput Phys Comm 2017;210:29–44. http://dx.doi.org/10.1016/j.cpc.2016.09.009, URL: http://www.sciencedirect.com/science/article/pii/S0010465516302806.

[44] Ghaïtanellis A, Violeau D, Liu PLF, Viard T. SPH simulation of the 2007 Chehalis Lake landslide and subsequent tsunami. J Hydraul Res 2021;1–25. http://dx.doi.org/10.1080/00221686.2020.1844814, arXiv:https://doi.org/10.1080/00221686.2020.1844814.

[45] Gafton E, Rosswog S. A fast recursive coordinate bisection tree for neighbour search and gravity. Mon Not R Astron Soc 2011;418:770–81. http://dx.doi.org/10.1111/j.1365-2966.2011.19528.x.

[46] Hernquist L, Katz N. TREESPH: a unification of SPH with the hierarchical tree method. Astrophys J Suplement 1989;70(419). http://dx.doi.org/10.1086/191344.

[47] Cavelan A, Cabezón RM, Korndorfer JHM, Ciorba FM. Finding neighbors in a forest: a b-tree for smoothed particle hydrodynamics simulations. 2020, http://dx.doi.org/10.48550/arXiv.1910.02639.

[48] Green S. Particle simulation using CUDA. Technical Report, NVIDIA Corporation; 2007.

[49] Hérault A, Bilotta G, Dalrymple RA. Achieving the best accuracy in an SPH implementation. In: Proceedings of the 9th SPHERIC workshop. Paris; 2014, p. 134–9.

[50] Saikali E, Bilotta G, Hérault A, Zago V. Accuracy improvements for single precision implementations of the SPH method. Int J Comput Fluid Dyn 2020;34:774–87. http://dx.doi.org/10.1080/10618562.2020.1836357, arXiv:https://doi.org/10.1080/10618562.2020.1836357.

[51] Hérault A, Bilotta G, Vicari A, Rustico E, Del Negro C. Numerical simulation of lava flow using a GPU SPH model. Ann Geophys 2011;54:600–20. http://dx.doi.org/10.4401/ag-5343.

[52] Ferrari A, Dumbser M, Toro EF, Armanini A. A new 3D parallel SPH scheme for free surface flows. Comput & Fluids 2009;38:1203–17. http://dx.doi.org/10.1016/j.compfluid.2008.11.012.
[53] Mayrhofer A, Rogers BD, Violeau D, Ferrand M. Investigation of wall bounded flows using SPH and the unified semi-analytical wall boundary conditions. Comput Phys Comm 2013;184:2515–27. http://dx.doi.org/10.1016/j.cpc.2013.07.004, URL: http://www.sciencedirect.com/science/article/pii/S0010465513002324.
[54] Brezzi F, Pitkäranta J. On the stabilization of finite element approximations of the Stokes equations. In: Hackbusch W, editor. Efficient solutions of elliptic systems: proceedings of a GAMM-seminar kiel, January 27 to 29 1984. Wiesbaden: Vieweg+Teubner Verlag; 1984, p. 11–9. http://dx.doi.org/10.1007/978-3-663-14169-3_2.
[55] Bingham EC. Fluidity and plasticity. New York: McGraw-Hill; 1922.
[56] Papanastasiou TC. Flows of materials with yield. J Rheol 1987;31:385–404. http://dx.doi.org/10.1122/1.549926.
[57] Ostwald W. Über die geschwindigkeitsfunktion der viskosität disperser systeme. Kolloid Zeitschrift 1925;36:99–117. http://dx.doi.org/10.1007/BF01431449.
[58] Herschel WH, Bulkley R. Konsistenzmessungen von gummi-benzol-losungen. Kolloid Zeitschrift 1926;39:291–300. http://dx.doi.org/10.1007/BF01432034.
[59] Alexandrou AN, McGilvreay TM, Burgos G. Steady Herschel–Bulkley fluid flow in three-dimensional expansions. J Non-Newton Fluid Mech 2001;100:77–96. http://dx.doi.org/10.1016/50377-0257(01)00127-6.
[60] De Kee D, Turcotte G. Viscosity of biomaterials. Chem Eng Commun 1980;6:273–82. http://dx.doi.org/10.1080/00986448008912535.
[61] Zhu H, Kim YD, De Kee D. Non-newtonian fluids with a yield stress. J Non-Newton Fluid Mech 2005;129:177–81. http://dx.doi.org/10.1016/j.jnnfm.2005.06.001.
[62] Farhadi A, Mayrhofer A, Tritthart M, Glas M, Habersack H. Accuracy and comparison of standard κ-ϵ with two variants of κ-ω turbulence models in fluvial applications. Eng Appl Comput Fluid Mech 2018;12:216–35. http://dx.doi.org/10.1080/19942060.2017.1393006.
[63] Rogers BD, Dalrymple RA. Three-dimensional SPH-SPS modeling of wave breaking. In: Symposium on ocean wave measurements and analysis. ASCE, Madrid; 2005.
[64] Español P, Revenga M. Smoothed dissipative particle dynamics. Phys Rev E 2003;67.
[65] Zago V, Bilotta G, Cappello A, Dalrymple RA, Fortuna L, Ganci G, Hérault A, Del Negro C. Preliminary validation of lava benchmark tests on the GPUSPH particle engine. Ann Geophys 2019;62. http://dx.doi.org/10.4401/ag-7870.
[66] Zago V, Almashan N, Dalrymple RA, Bilotta G, Al-Houti DB, Neelamani S. Validation of an SPH-FEM model for offshore structure. In: Proceedings of the 16th SPHERIC workshop. Catania; 2022.
[67] Schulze LJ, Zago V, Bilotta G, Dalrymple RA. Localized kernel gradient correction for SPH simulations of water wave propagation. In: Proceedings of the 16th SPHERIC workshop. Catania; 2022.
[68] Clang Team, a. clang: a c language family frontend for LLVM. URL: https://clang.llvm.org/.
[69] Wu J, Belevich A, Bendersky E, Heffernan M, Leary C, Pienaar J, Roune B, Springer R, Weng X, Hundt R. Gpucc: An open-source GPGPU compiler. In: Proceedings of the 2016 international symposium on code generation and optimization. New York, NY, USA: Association for Computing Machinery; 2016, p. 105–16. http://dx.doi.org/10.1145/2854038.2854041.
[70] Clang Team, b. compiling CUDA with clang. URL: https://llvm.org/docs/CompileCudaWithLLVM.html.
[71] G. Bilotta. Bug 52037 - performance regression in CUDA code compiled by clang 13 versus clang 12 (clang 12 ~50% faster than clang 13). URL: https://bugs.llvm.org/show_bug.cgi?.
[72] Volkov V. Better performance at lower occupancy. In: Proceedings of the GPU technology conference. vol. 10, 2010.
[73] Volkov V. Understanding latency hiding on gPUs. (Ph.D. thesis), Berkeley: EECS Department, University of California; 2016.
[74] Domínguez JM, Fourtakas G, Altomare C, Canelas RB, Tafuni A, García-Feal O, Martínez-Estévez I, Mokos A, Vacondio R, Crespo AJC, Rogers BD, Stansby PK, Gómez-Gesteira M. DualSPHysics: from fluid dynamics to multiphysics problems. Comput Part Mech 2022;9:867–95. http://dx.doi.org/10.1007/s40571-021-00404-2.
[75] Bilotta G, Zago V, Centorrino V, Dalrymple RA, Hérault A, Del Negro C. A numerically robust, parallel-friendly variant of BiCGSTAB for the semi-implicit integration of the viscous term in smoothed particle hydrodynamics. J Comput Phys 2022;446:111413. http://dx.doi.org/10.1016/j.jcp.2022.111413, URL: https://www.sciencedirect.com/science/article/pii/S0021999122004752.