

We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900

Open access books available

186,000

International authors and editors

200M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Design and Implementation of Particle Systems for Meshfree Methods with High Performance

Giuseppe Bilotta, Vito Zago and Alexis Hérault

Abstract

Particle systems, commonly associated with computer graphics, animation, and video games, are an essential component in the implementation of numerical methods ranging from the meshfree methods for computational fluid dynamics and related applications (e.g., smoothed particle hydrodynamics, SPH) to minimization methods for arbitrary problems (e.g., particle swarm optimization, PSO). These methods are frequently embarrassingly parallel in nature, making them a natural fit for implementation on massively parallel computational hardware such as modern graphics processing units (GPUs). However, naive implementations fail to fully exploit the capabilities of this hardware. We present practical solutions to the challenges faced in the efficient parallel implementation of these particle systems, with a focus on performance, robustness, and flexibility. The techniques are illustrated through GPUSPH, the first implementation of SPH to run completely on GPU, and currently supporting multi-GPU clusters, uniform precision independent of domain size, and multiple SPH formulations.

Keywords: software design, particle systems, SPH, GPGPU, high-performance computing, numerical stability, best practices

1. Introduction

Particle systems were first formally introduced in computer science by Reeves [1], as the technique used by Lucasfilm Ltd. for the realization of some of the special effects present in the film *Star Trek II: The Wrath of Khan* [2]. Since then, particle systems have been used in computer graphics for the simulation of visually realistic fire, moving water, clouds, dust, lava, and snow. A particle system consists of a collection of distinct elements (particles) that are generated according to specific rules, evolve and move in the simulation space, and die out at the end of their life cycle. The position and characteristics of the particles in the system over simulated time are then used to render larger bodies (flames, rivers, etc.) with appropriate techniques [3].

While originally developed purely for visual effects, and associated with evolution laws focused on the final appearance rather than the physical correctness of the behavior, particle systems also form the digital infrastructure for the implementation of a class of numerical methods (known as meshless, meshfree, or particle methods) that have started emerging since the late 1970s as alternatives to the

traditional grid-based numerical methods (finite differences, finite volumes, finite elements). These methods—smoothed particle hydrodynamics (SPH) [4], reproducing kernel particle method (RKPM) [5], finite pointset method (FPM) [6], discrete element method (DEM) [7], etc.—provide rigorous methods to discretize the physical laws governing the continuum and thus provide physics-based evolution law for the properties of the particles that act both as interpolation nodes in the mathematical sense and as virtual volumes of infinitesimal size carrying the properties of the macroscopic mass they represent.

More recently, the same computer techniques have been used to solve more abstract problems. For example, particle swarm optimization (PSO) [8] is a methodology to find approximate minima for functions whose derivatives cannot be computed (at all or in reasonable times), in spaces of arbitrary dimensions. Outside of the particle systems in the proper sense, simulation methods such as molecular dynamics (MD) [9] and many large-scale agent-based models present significant similarities with particle systems [10, 11] and share much of the infrastructural work with it.

Particle systems are computationally intensive. Realistic visual effects, accurate physical simulations, fast minimization, and large-scale agent-based models all require thousands if not millions (or more) of particles. On the upside, the behavior of most particle systems can be described in an embarrassingly parallel way, where each particle evolves either independently from the rest of the system or with at most local knowledge of the state of the system. This property makes particle systems a perfect fit for implementation on massively parallel computational hardware following the stream processing programming model, and in particular modern graphics processing units (GPUs), that have grown in the last decade from fast, programmable 3D rendering hardware to more general-purpose computing accelerators [12].

While the parallel computational power of GPUs is a natural fit for the parallel nature of particle systems, naive implementations will miss many opportunities to fully exploit GPUs, even when achieving performance orders of magnitude higher than an unoptimized, serial CPU implementation. Our objective is to discuss the optimal implementation of particle systems on GPU, so that anyone setting forth to implement a particle system can draw from our experience to avoid common pitfalls and be aware of the implications of many design choices. Optimality will be viewed in terms of performance (achieving the highest number of iterations per second in the evolution of the system), robustness (numerical stability), and flexibility (allowing the implementation of a wide range of variants for the particle system, e.g., to allow the simulation of different phenomena).

We will show that while these objectives are sometimes in conflict—so that the developer will have to choose to, e.g., sacrifice performance for better numerical stability—there are also cases where they complement each other, e.g., with some numerically more robust approaches also being more computationally efficient or with certain design choices for the host code structure being also more favorable to future extensions to multi-GPU support.

We will make extensive reference to our experience from the implementation of GPUSPH [13–17], the first implementation of SPH to run completely on GPU using CUDA and currently supporting multi-GPU and multi-node distribution of the computation. However, all the themes that we discuss and solutions we present are of interest to all particle systems and related methods, regardless of the specific theoretical background underlying them. To show this, simpler examples to illustrate the benefits of individual topics discussed will also be presented through a reduced implementation of PSO. Some of the most advanced techniques described can be seen in action in GPUSPH itself, which is freely available under the GNU General Public License, version 3 or higher [18].

While our focus will be on GPU implementation, many of the approaches we discuss can bring significant benefits even on CPU implementations, allowing better exploitation of the vector hardware and multiple cores of current hardware. The intention is thus to provide material that is of practical use regardless of the specific application and hardware.

2. Terminology and notation

Throughout the paper, we will rely on the terminology used by the cross-platform OpenCL standard [19]. All the concepts we discuss will be equally valid in different programming contexts, such as the proprietary CUDA developed by NVIDIA specifically for their GPU and HPC solutions [20]. This choice stems from the authors' opinion that the OpenCL terminology is more neutral and less susceptible to the kind of confusion that some vendors have leveraged as a marketing tactic in promoting their solutions.

In our examples, we will also frequently refer to “small vector” data types. These are types in the form `typeN` where `type` is a primitive type (such as `char`, `int`, `float`, `double`) and `N` is one of 1, 2, 3, 4, 8, and 16. For example, a `float4` would be a structure that in C or C++ could be defined as `struct float4 {float x, y, z, w;}`. Following OpenCL, the components of the small vector types will be named `x`, `y`, `z`, `w` for types with up to 4 components, and `s0`, ... `s9`, `sa`, ... `sf` for types with up to 16 components. In some examples we also make use of the OpenCL “swizzle notation,” such that, for example, given `float2 v=(0.0f, 1.0f);`, then `v.xxyy` is a `float4` with components `(0.0f, 0.0f, 1.0f, 1.0f)`.

We will assume that each small vector type is “naturally aligned,” when `N` is a power of two: a `typeN` will begin at a memory address which is a multiple of `N*sizeof(type)`; for `N=3`, we will assume that `type3` begins at a memory address which is a multiple of `sizeof(type)`. This is in contrast to OpenCL, whose `cl_type3` types are assumed to be aligned like the corresponding `cl_type4` types, and special `vload3` instructions are needed to load packed 3-vectors. We will also show momentarily that such 3-component types should in general be avoided as they lead to lower performance, since most if not all modern hardware are designed around power-of-two types (which is the reason why the OpenCL type is aligned to four components).

Finally, we will assume that the standard operations (component-by-component addition, subtraction, and multiplication, multiplication by a scalar, dot product) on the small vector types have been defined, in the usual manner. (OpenCL C defines these as part of the language, for CUDA appropriate overloads for the common operators must be defined by the user.)

3. The GPU programming model

3.1 Stream processing

Modern GPUs are designed around the *stream processing* paradigm, a simplified model for shared-memory parallel programming that sacrifices inter-unit communication in favor of higher efficiency and scalability.

At an abstract level, stream processing is defined by a sequence of instructions (a *computational kernel*) to be executed on each element of an input data stream to produce an output data stream, under the assumption that each input element can be processed independently from the others (and thus in parallel). A computational kernel is similar to a standard function in classic imperative programming

languages; at runtime, as many instances of the function will be executed as necessary to cover the whole input data stream. Such instances (*work-items*) may be dispatched in concurrent batches, running in parallel as far as the hardware allows, and the programmer is generally given very little control, if any, on the dispatch itself, other than being able to specify how many instances are needed in total. This choice allows the same kernel to be executed on the same data stream, adapting naturally to the characteristics of the underlying hardware, and is one of the main characteristics of stream processing.

For example, if the hardware can run 1000 concurrent work-items, but the input stream consists of 2000,000 total elements, the hardware may batch 1000 work-items for execution at once and then dispatch another 1000 when the first batch completes execution. This continues until the entire input stream has been processed, executing 2000 total batches. For the same workloads, more powerful hardware able to run 100,000 concurrent work-items may be able to complete sooner by issuing 20 total batches, in a manner completely transparent to the programmer.

This programming model fits very well the simpler workload needed in many steps of the rendering process for which GPUs are designed: in such a case, the input and output streams may consist of the data and attributes for the vertices in the geometries describing the scene, for example, or for the fragments produced by the rasterization of such geometries. However, the more sophisticated requirements of general-purpose programming have led to the extension of the stream processing paradigm to provide programmers with finer control on the work-item dispatch as well as the possibility for efficient data sharing between work-items under appropriate conditions.

A modern stream processing *device* (typically a GPU, but may also be a multicore CPU with vector units, a dedicated accelerator like Intel's Xeon Phi, or a special-design FPGA) is composed of one or more *compute units* (each being a CPU core, a GPU multiprocessor, etc.) equipped with one or more *processing elements* (a SIMD lane on CPU, a single stream processor on GPU, etc.), which are the hardware components that process the individual work-items during a kernel execution. The programming model of these devices, as presented, e.g., by standards such as OpenCL [19] and by proprietary solutions such as NVIDIA CUDA [20], exposes the underlying hardware structure by allowing the programmer to specify the granularity at which work-items should be dispatched: each *workgroup* is a collection of work-items that are guaranteed to run on a single compute unit; work-items within the same workgroup can share data efficiently through dedicated (often on-chip) memory and can synchronize with each other, ensuring correct instruction ordering. Tuning workgroup size and the way work-items in the same workgroup access data can have a significant impact on performance.

The GPU multiprocessors are further characterized by an additional level of work-item grouping at the hardware level, as the work-items running on a single multiprocessor are not independent from each other: instead, a single instruction pointer is shared by a fixed-width group of work-items, known as the *warp* on NVIDIA GPUs, or *wavefront* on AMD GPUs, corresponding in a very general sense to the vector width of SIMD instructions on modern CPUs. We will use the hardware-independent term *subgroup* (as introduced, e.g., in OpenCL 2.0) to denote this hardware grouping. The subgroup structure of kernel execution influences performance in a number of ways. The most obvious way is that the size of a workgroup should always be a multiple of the subgroup size: a partial subgroup would be fully dispatched anyway, but masked, leading to lower hardware usage. Additional aspects where the subgroup partitioning can influence performance are branch divergence and coalesced memory access.

Branch divergence occurs when work-items belonging to the same subgroup need to take different execution paths at a given conditional. Since the subgroup proceeds in lockstep for all intents and purposes, in such a situation the hardware must mask the work-items not satisfying either branch, execute one side of the branch, invert the mask, and execute the other side of the branch: the total runtime cost is then the sum of the runtimes of each branch. If the work-items taking different execution paths belong to separate subgroups, this cost is not incurred, because separate subgroups can execute concurrently on different code paths, leading to an overall runtime cost equal to that of the longer branch.

Coalescence in memory access is achieved when the controller of a GPU can provide data for the entire subgroup with a single memory transaction. Ensuring that this happens is one of the primary aspects of efficient GPU implementations and will be the basis for many of the performance hints discussed later on.

3.2 Stream processing and particle systems

Stream processing is a natural fit for the implementation of particle systems, since the vast majority of algorithms that rely on particle systems are embarrassingly parallel in nature, with the behavior of each particle determined independently, thus providing a natural map between particles and work-items for most kernels. This allows naive implementations of particle systems to be developed very quickly, often with massive performance gains over trivial serial implementations running on single-core CPUs.

Such implementations will however generally fail at leveraging the full computational power of GPUs, except in the simplest of cases. Any moderately sophisticated algorithm will frequently require a violation of the natural mapping of particles to stream elements (and thus work-items), either in terms of data structure and access or in terms of implementation logic, to be able to achieve the optimal performance on any given hardware.

3.3 Limitations in the use of GPUs

Programmable GPUs have brought forth a revolution in computing, making (certain forms of) large-scale parallel computing accessible to the masses. Many applications have seen significant benefit from a transition to the GPU as supporting hardware, and in response vendors have improved GPU architectures, making it easier to achieve better performance with less implementation effort.

When choosing the GPU as preferential target platform, however, developers must take into consideration the fact that not all users may have high-end professional GPUs, and while the stream computing paradigm is largely sufficient in compensating for the difference in computational power, there are at least two significant aspects that must be explicitly handled.

Memory amount is one of these issues: consumer GPUs typically only have a fraction of the total amount of RAM offered in professional or compute-dedicated devices: while the latter may feature up to 16GB of RAM, low-end devices may have 1/4th or even 1/8th of that. Moreover, even the amount of memory available on high-end devices may be insufficient to handle larger problems. Software should therefore be designed to allow distribution of computation across multiple devices.

The second issue is that, being designed for computer graphics, GPUs typically focus on single-precision (32-bit) floating-point operations, and double precision (64-bit) may be either not supported at all or supported at a much lower execution rate (as low as 1:32) than single precision, which may remove the computational advantage of using GPUs in the first place (this can be true even on high-end GPUs,

as was infamously the case for the Maxwell-class Tesla GPUs from NVIDIA). Designing the software around the use of single precision can therefore allow supporting higher performance across a wider class of devices, but it may require particular care in the handling of essential state variables in particle systems. This will be discussed in Section 5.

4. Performance

While GPUs provide impressive computational power compared to CPUs, this is offset by a much higher sensitivity to data layout and access patterns: even a very computationally intensive kernel may result memory bound if the appropriate care is not given to these aspects.

The main GPU memory (*global* memory) is characterized by having high bandwidth, but also very high latency: access to global memory may consume hundreds of cycles, and work-items waiting for data will not proceed until the data is available to all of them, at least at the subgroup granularity.

Under appropriate conditions (called memory coalescing or fast-path), the GPU can provide data for a whole subgroup with a single memory transaction. Optimal access patterns in this regard are achieved when the work-items in a subgroup request data which is consecutive in memory, properly aligned (i.e., with the lowest-index element starting at an address which is a multiple of the data size times the subgroup size), and with specific size constraints—typically power-of-two sizes, up to 128 bits per work-item: essentially, the equivalent of a float, float2, or float4, but not float3.

When fast-path requirements are not satisfied, the impact on kernel run times can be dramatic, especially on older architectures: designing data structures and algorithms around these requirements is therefore one of the main topics we will address. But even when coalesced access is achieved, each subgroup will have to wait for at least one memory transaction before proceeding to the instruction that makes use of the data. To hide this latency, GPU multiprocessors are designed to keep multiple workgroups alive concurrently and will automatically switch to another active workgroup (or subgroup within the same workgroup), while one is stalled waiting for data; to make efficient use of this capability, it is necessary to *overcommit* the device, i.e., issue kernels with more workgroups than would theoretically be able to run concurrently on the given hardware.

For example, on a GPU with 16 multiprocessors, each equipped with 128 streaming processors, it will not be sufficient to issue kernels with 2048 work-items to fully exploit the hardware: to fully hide operation latency, the developer should aim for global work sizes which are at least an order of magnitude more than the bare minimum.

A GPU that is fully under load is said to be *saturated*. On most modern architectures, tens of thousands of work-items are generally necessary to saturate mid- and high-end devices. This condition is usually satisfied for any moderate or large particle system, in which case the data layout and access patterns become the bottleneck for memory bandwidth utilization.

4.1 Array of structures versus structure of array

The first step in improving GPU bandwidth usage is to avoid high-level structured data layouts and store information about the particle system in a “transposed” format.

Let us consider, for example, a simple particle system in three dimensions, where each particle is described by its position (3 floats) and velocity (3 floats). In a CPU implementation, data would be stored in a format based on a `Particle` structure, and the particle system would be an array of `Particles`. Integrating the particles' position over a time-step dt would be achieved in a simple loop like the one illustrated in **Listing 1**.

This approach is called *array of structures* (AoS), and assuming a stream processing perspective, preserving the same layout would lead to a compute kernel in the form presented on the left in **Listing 2**. However, since each particle is more than 128 bit wide, the GPU would not be able to satisfy each subgroup access to the `particle_system` (marked by the comments) in a single transaction, resulting in a potential slowdown of an order of magnitude or more. A better solution on GPU would be to split the particle structure in each primary component and thus have, in this case, an array of positions and an array of velocities, as shown on the right in **Listing 2**.

Part of the advantage of this approach (*structure of arrays*, SoA) is the natural higher access granularity, which limits read and write access to what is strictly necessary. With the AoS approach, it is also possible to limit writes to the specific parts, e.g., `particle_system[i].pos+= particle_system[i].vel*dt`, but we will see that this only partially recovers the performance gap against SoA. Moreover, the access granularity of SoA also reflects in the function signatures, improving developer discipline. The downside is the growing number of buffers, and strategies to manage this will be discussed in Section 6.2.1.

Listing 1.
Simple host code to integrate the position of a particle system.

<pre>struct Particle { float3 pos; float3 vel; };</pre>	<pre>void integrate_pos(Particle *particle_system, size_t N, float dt) { for (size_t i=0; i<num_particles; ++i) { Particle& p=particle_system[i]; p.pos+= p.vel*dt; } }</pre>
---	--

Listing 2.
Particle system with stream processing: array of structure (left) versus structure of array (right).

<pre>kernel void integrate_pos(Particle *particle_system, size_t N, float dt) { size_t i=get_global_id(0); if (i>= N) return; /* read the old particle state */ Particle p=particle_system[i]; p.pos+= p.vel*dt; /* write the new particle state */ particle_system[i]=p; }</pre>	<pre>kernel void integrate_pos(float3 *posArray, const float3 *velArray, size_t N, float dt) { size_t i=get_global_id(0); if (i>= N) return; float3 pos=posArray[i]; const float3 vel=velArray[i]; pos+= vel*dt; posArray[i]=pos; }</pre>
--	--

Further optimizations, particularly important on older architectures, can be achieved with the sacrifice of some memory, to provide the position and velocity with a fourth (unused) component, as illustrated in **Listing 3** (left), resulting in better bandwidth usage and thus faster execution; moreover, additional frequently used data may be stored in the fourth component, if needed (e.g., in GPUSPH we store the mass in `pos.w` and the density in `vel.w`).

The benefits of the discussed strategies are exemplified in **Table 1** for a simple three-dimensional implementation of PSO. The specific values will obviously generally depend on the specific compute kernel as well as on the specific hardware.

Some additional (usually minor) benefits can be achieved by explicitly telling the compiler that the position and velocity array never intersect; this is achieved using the `restrict` specification for the pointer (**Listing 3**, right) which, for more complex kernels, may allow the compiler to produce faster code by assuming no dependencies between writes on one array and reads on the other. On some hardware, `const * restrict` arrays are also made accessible through a dedicated cache, further improving performance.

4.2 Wide arrays

The SoA approach can provide a significant boost in performance on GPU, as long as the individual parts of the structure (position, velocity, etc.) fit within the size requirements for coalesced memory access. When even individual structure members are wider than the optimal 128-bit width, however, alternative approaches are necessary. An example of this occurrence is the storage of a list of

Listing 3.
Using efficient data types on GPU (left) and leveraging the power of restricted pointers (right).

<pre>kernel void integrate_pos(float4 *posArray, const float4 *velArray, size_t N, float dt) { size_t i=get_global_id(0); if (i>= N) return; float4 pos=posArray[i]; const float4 vel=velArray[i]; pos.xyz+= vel.xyz*dt; /* OpenCL swizzle */ posArray[i]=pos; }</pre>	<pre>kernel void integrate_pos(float4 * restrict posArray, const float4 * restrict velArray, size_t N, float dt) { size_t i=get_global_id(0); if (i>= N) return; float4 pos=posArray[i]; const float4 vel=velArray[i]; pos.xyz+= vel.xyz*dt; /* OpenCL swizzle */ posArray[i]=pos; }</pre>
---	---

	AoS	Selective AoS	SoA	Padded SoA
Runtime (ms)	98	73	25	13
Speedup (prev)	—	1.3	2.9	1.9
Speedup (total)	—	1.3	3.9	7.5
<i>2^24 particles running on an NVIDIA GeForce GT 750M.</i>				

Table 1.
Runtime comparison for a simple three-dimensional particle swarm optimization implementation, using the discussed paradigms: array of structures, array of structures with selective writing, structure of arrays, structure of arrays with padded members (i.e., using four instead of three components).

neighbors; frequently, the number of neighbors for a particle will range in the tens or hundreds, sometimes even more, requiring storage of as many integers per particle. Another example is given by particle systems with high dimensionality (higher than 4), which could arise, for example, for a particle swarm optimization approach to the minimization of the cost function of a deep neural network. The position (and velocities) of particles in such a system might require hundreds, thousands, or even more, components.

The optimal storage solution for the array holding the data in such cases is transposed compared to the natural order: whereas for most CPU code it is natural to first store the data belonging to the first particle, then the data belonging to the second particle, etc., the optimal GPU storage for these wide arrays is to first store the first component for each particle, followed by the second component for each particle, etc. Using the standard C array notation, the i -th component of the p -th particle in the classic format would be found at location $\text{data}[p \cdot \text{num_components} + i]$, whereas the optimal GPU location would use the addressing $\text{data}[i \cdot \text{num_particles} + p]$. Similarly, neighbors would be stored interleaved: the first neighbor of each particle, followed by the second neighbor for each particle, etc. This ensures that when particles iterate over their neighbors, they fetch the neighbor index in coalescence. The concept is illustrated in **Figure 1** (top and middle graphs).

The data transposition can rely on different chunk sizes; the single components approach discussed so far has the benefit of being simpler and the natural choice when each component needs to be treated independently (e.g., neighbors list traversal); if possible, however, wider chunks (e.g., using arrays of float2 or float4 elements instead of float) should be used (**Figure 1**, bottom), to achieve better utilization of the memory bandwidth.

In general, the balance between transposition and chunk width should be calibrated based on the hardware capability: current GPUs achieve optimal performance with float4s, while on a CPU or a Xeon Phi, the wide vector width offered by AVX and AVX-512 could be better exploited using float8 or float16 chunks, as illustrated in **Table 2**.

4.3 Particle sorting and neighbor search

In many particle systems, the behavior of the individual particles depends on the state of the particles in a neighborhood of the particle itself. The neighborhood may

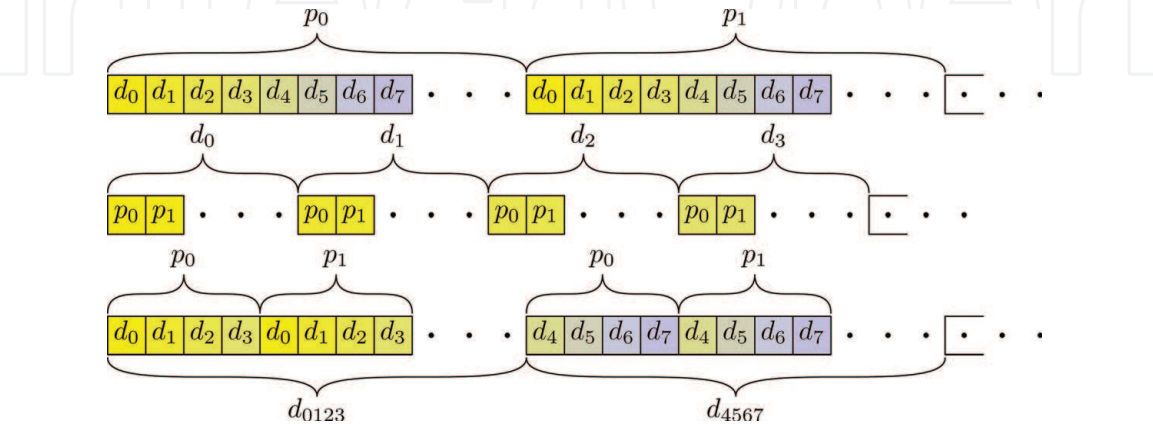


Figure 1. Possible memory layouts for wide arrays. Top, standard (particle-major) layout; middle, transposed (component-major) layout; bottom, transpose-chunked layout. Memory locations are colored by data component access: locations with the same color are accessed concurrently in parallel by all work-items. In the chunked case, more than one location may be accessed concurrently, depending on the chunk size and hardware capability.

Hardware	Naïve	Transposed	Chunk 2	Chunk 4	Chunk 8	Chunk 16
NVIDIA	476	41	39	38	48	71
Intel GPU	237	75	63	58	59	91
Intel CPU	120	568	294	422	53	41

The NVIDIA GPU is a GeForce GT 750M; the Intel GPU is an Intel HD Graphics Haswell GT2 Mobile, using the Beignet OpenCL implementation from Intel; the Intel CPU is an Intel Core i7-4712HQ, using the Intel OpenCL SDK 6.4.0.25. Bold italic values show the best performance. The CPU exhibits worse performance for the transposed layout than the naive due to the auto-vectorization introduced by the OpenCL implementation.

Table 2.

Median runtimes (in ms) of the position update kernel for a 128-dimensional particle swarm optimization using the described memory layouts, on different hardware.

be defined in terms of some fixed influence radius or may be determined dynamically, either based on a changing influence radius or based on a pure neighbors count (e.g., “the 10 closest neighbors”). Performance of particle systems on GPU can be improved by reordering particle data in memory so that the data for particles that are close to each other in the domain metric (e.g., distance) are also close in device memory, providing more opportunities for coalesced memory access and (when available) better cache utilization [21].

Sorting is generally achieved using key/value pairs, with the particle hash key computed from the particle position in space: the key array is then sorted, and all data arrays are reordered based on the new key array positions. Common ways to compute the particle sort key are based on either n-trees [22] or regular grids [23]. The main advantage of using an n-tree (and thus in particular quadtrees in 2D and octrees in 3D) is the adaptive nature of the structure, which is denser where particles are concentrated and sparser in the domain regions where particles are more spread out. By contrast, regular grids result in cells which are uniformly spaced and thus in unbalanced particle distributions among the cells.

The adaptive nature of n-trees can result in performance gains in a number of use cases, such as nearest-neighbor searches, collision detection, clump finding, and rendering. At the same time, traversing the tree structure itself efficiently on a stream processing architecture is nontrivial and often results in sub-optimal memory bandwidth utilization [24]. Regular grids, on the other hand, have a much simpler and computationally less expensive implementation, they lead to efficient neighbor search with fixed radius (as we will discuss momentarily), and the resulting data structures can also be used to support domain decomposition in the multi-GPU case, as we will discuss in Section 4.5.3, and also to improve the numerical robustness of the particle system, as we will discuss in Section 5.4.

4.3.1 Regular grids for neighbor search

Given a neighbor search radius r , we can subdivide the domain with a regular grid where the stepping in each direction is no less than r . This guarantees that the neighbors for any particle in any given cell can only be found at most in the adjacent cells in each of the cardinal and diagonal directions (Moore neighborhood of radius 1), as depicted in **Figure 2**.

We can then sort particles (i.e., their data) by, e.g., the linear index or the Morton code [25] of the cell they belong to (computed from the particle global position), so that data for all particles belonging to one cell ends up in a consecutive memory region. Furthermore, we can store in a separate array the offset (common

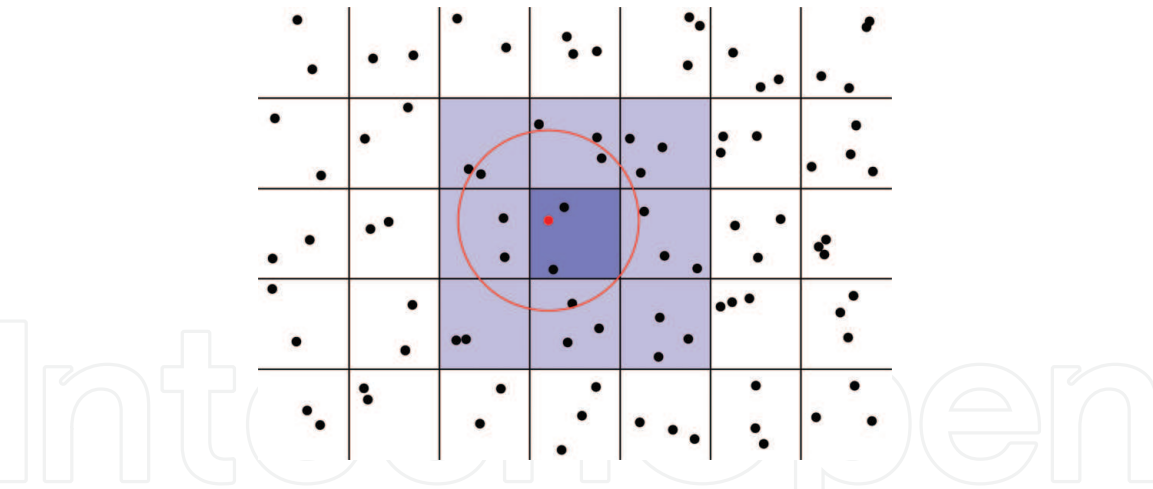


Figure 2.
Support grid for neighbor search: if the cell side is no less than the influence radius, the neighbors for any particle in any given cell (dark blue square in the picture) can be found in at most the Moore neighborhood of the cell itself (light blue squares in the picture).

to all data arrays) to the beginning of the data for the particles in each cell (Figure 3).

A single particle can then search for neighbors by only looking at the corresponding subsets of the particle system, starting from the cell start index for each adjacent cell. Since all particles belonging in the same cell will need to traverse the same subset of the particle list, further improvements can be obtained by loading the data about the potential neighbors into a shared-memory array.

4.3.2 Just-in-time neighbor search versus neighbor list storage

The results of the neighbor search may be used immediately (e.g., by computing the particle-particle interaction as each neighbor is found) or deferred: in the latter case, the neighbor search itself constitutes its own step in the system evolution, and the results of the search are stored in a list which is then used in subsequent kernels when particle-particle interactions must be computed.

The “just-in-time” approach (which can be equivalently seen as searching for neighbors whenever needed) has the advantage of lower memory requirements (since the list of neighbors needs not be stored), but the disadvantage that the cost of the search itself must be paid whenever interactions must be computed. Therefore, it is the preferred approach when the results of each search are only needed once. Conversely, when the results of the neighbor search are to be used multiple times, it is better, performance-wise, to store the results, and then reuse them in the

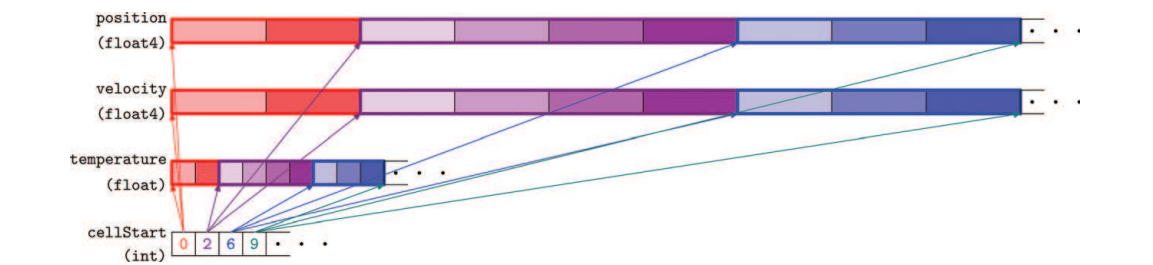


Figure 3.
Memory layout for the support grid. Data arrays are sorted so that the data belonging to particles in any given cell is consecutive in memory, and a separate array holds the offset to the beginning of the data of the particles in each cell. In the picture, each primary color denotes a cell, and the color gradient refers to different particle data within each cell. The first cell has two particles, the second has four particles, and the third has three.

following steps, in order to amortize the cost of the search. The downside in this case is much higher memory requirements.

For example, in GPUSPH the cost of the particle sorting and neighbor list construction can take as much as 30% of the runtime of a single step; most of this (25% of the step runtime) is spent in the neighbor search phase of the neighbor list construction; the list itself is then used for all subsequent kernels that require particle-particle interaction (boundary conditions, density smoothing, forces computation, surface detection, etc., most of which are executed twice due to the predictor/corrector integration scheme used).

Working without a neighbor list, the runtime cost of the search would have to be paid for each execution of a kernel with particle-particle interaction, increasing the runtime of a single operation by over 50%. The neighbor list is therefore a better choice for performance.

On the other hand, on a typical simulation in GPUSPH, the neighbor list is also responsible for the highest memory allocation, even including the double buffering required for most data arrays: indeed, all of the particle properties combined take between 100 and 300 bytes per particle (depending on the formulation being used), whereas for the neighbor list, we need to store, in the simplest cases, 128 neighbors, leading to an occupation of 512 bytes per particle. Some formulations may require two or three times as many neighbors per particle.

We remark that even when not all particles have the exact same number of neighbors, the neighbor list should be statically allocated at the beginning of the simulation, with the capacity to hold the maximum (expected) number of neighbors for any particle and with the layout discussed in Section 4.2, to maximize the performance of its traversal. This leads to some potential memory waste for the benefit of performance. However, even a more conservative and less wasteful neighbor list would still occupy significant memory (e.g., a median of 80 neighbors per particle still needs to be tracked on a typical GPUSPH simulation, which only lowers the memory occupation for the neighbor list to 320 bytes per particle). The net result is that the large allocations required by the neighbor list will limit the maximum size of a particle system that can be simulated on a single GPU.

Additionally, even with the stored neighbor list, nearly one-third of a simulation step is still taken by its construction. A solution to this issue is to only update the list every n iterations, with n large enough to reduce the performance impact and small enough to not affect the results in a significant way. With the default choice of $n = 10$ in GPUSPH, the cost of particle sorting and neighbor search drops to around 5% of the total runtime in the worst cases. To improve the reliability of the simulations when neighbor list updates are less frequent, a good strategy is to increase the search radius: with this approach, given an influence radius r (maximum distance for interaction), the neighbors are actually added to the list of neighbors if their distance from the central particle is less than αr , with $\alpha > 1$; neighbors whose distance from the central particle is larger than r are then skipped in the kernels. The larger search radius takes into account the fact that particles may move before the next neighbor list update, thus bringing them closer. In this sense, the expansion factor for the neighbor search should be computed based on $nv_M\Delta t$, where v_M is the maximum expected (relative) particle velocity and Δt the maximum expected time step.

4.4 Heterogeneous particle systems

While simple particle systems are often homogeneous (in that all particles behave the same way), many applications require heterogeneous particle systems,

where particles behave differently depending on some intrinsic characteristic. For example, SPH for fluid dynamics typically needs at least two different particle types: fluid particles that track the fluid itself and boundary particles that define solid walls, moving objects, etc.; the way particles interact with each other (or even whether or not they interact at all) in this case depends on both the central and neighboring particle type.

Heterogeneity in the behavior of the particles and their interactions can have a significant impact on the performance of the system, particularly when it is stored together, without any specific attention to the distribution of the particles and their types. Indeed, the natural way to process a particle system is to issue, for most kernels, a work-item per particle. However, when particles with different types or behavior are processed by work-items in the same subgroup, this leads to divergence, slowing down execution. Similarly, when particles iterate over neighbors, they may have neighbors of different types at corresponding indices (e.g., the third neighbor of the first particle may be a fluid neighbor, while the third neighbor of the second particle might be a boundary neighbor); in this case, again, kernel execution will incur divergences, even if the central particles are of the same type. Moreover, since the distinction between particle types and interaction form must be done at kernel runtime, the kernel code itself grows more complex, reducing optimization opportunities for the compiler and leading to sub-optimal usage of private variables, frequently resulting in register spills, where a reserved area of global memory gets used for temporary storage of private work-item variables, with a severe impact on performance.

When the heterogeneous behavior is due to some static property (e.g., a fixed particle type property), the most obvious choice is to split the particle system itself (e.g., have a particle system for fluid particles and a separate particle system for boundary particles). This has several advantages: it is possible to run a kernel on particles of a specific type more efficiently while still making it possible to run a kernel on all particles; it is also possible to do selective allocations, for example, if a given property (e.g., object number) is only needed for a specific type. The downside is that the management code becomes more complex, and kernels where particles of one type need to interact with particles of the other type must be given access to both sets of arrays, which can increase the complexity of the kernel signatures.

A simpler approach that does not completely solve the divergence issue but can greatly reduce the occurrences of divergence is to introduce additional sorting criteria. For example, one can sort particles by cell, and within cell then sort particles by type, so that for any cell one finds first all the fluid particles (in that cell), followed by all the boundary particles (in the same cell). This “specialized sorting” approach reduces the occurrences of subgroups spanning multiple types to those crossing the boundary between types or between cells. In GPUSPH, the introduction of the per-type sorting within cells has improved the performance of the particle-particle interaction by around 2%. A significant advantage of this approach compared to the split system mentioned before is that it can be used also when the criteria for the behavioral difference are dynamic.

4.4.1 Split neighbor list

Divergence during the neighbor list traversals can be avoided by using a split neighbor list that separately stores neighbors of each type. This comes naturally when using separate particle systems and is efficient also with the specialized sorting approach, since potential neighbors of the same type will be enumerated

consecutively in each cell. The split neighbor list can be implemented in such a way that it is possible to iterate efficiently on neighbors of only one given type (or otherwise satisfying one given splitting criterion) while still preserving the possibility to iterate over all neighbors when necessary and minimizing allocation.

A naive split neighbor list can be implemented with separate allocations (e.g., a separate neighbor list per type), but this can quickly lead to an explosion of the already significant memory usage due to the existence of the neighbor list itself: without additional information on the neighbor distribution by type, for example, it may be necessary to allocate a full-sized neighbor list for each type. A more compact solution without loss of traversal efficiency can be achieved by storing the split neighbor list in a single allocation but filling the per-type section of the list from different ends.

As an example, consider the case of two particle types (fluid and boundary), and assume that the neighbor list can hold M neighbors per particle (M is the maximum number of neighbors any particle can have). For each particle, we store the fluid neighbors starting from position 0 (using the 0-based indexing common in the C language family), moving forward, and the boundary neighbors starting from position $M-1$, backward, where the indices refer to the particle-specific section of the full neighbor list, and taking interleaving into account as described in Section 4.2; iterating over all fluid neighbors is then achieved in the usual way, whereas iterating over all boundary neighbors would be achieved by traversing the array in reverse, as illustrated in **Listing 4**.

To prevent one section of the neighbor list from bleeding into the other, it is now necessary to put a marker (i.e., an index with a special value, such as -1 , defined as `NEIBS_END` in the example in **Listing 4**) at the end of each of the sides of the list, which implies that the effective maximum number of neighbors is reduced by 1 (with the end-of-list marker shared between the two sides when the neighbors list is otherwise full).

If there are more than two types of particle, the same strategy can still be applied, by sectioning the neighbor list. For example, with four types A, B, C, and D, we need to set a value $M1$ (the total number of neighbors of type A and B combined) and $M2$ (the total number of neighbors of type C and D combined); the neighbor list is allocated to hold $M1+M2$ neighbors per particle, where neighbors of type A are stored from position 0 onward, neighbors of type B are stored from position $M1-1$ backward, neighbors of type C are stored from position $M1$ onward, and neighbors of type D are stored from position $M1+M2-1$ backward. Type pairs should be chosen, when possible, based on the uniformity of the cumulative number of neighbors (e.g., if it is more likely that the sum of A and C neighbors is constant, it is better to pair A with C than with B).

Listing 4.

Traversing the split neighbors list: first type (fluid particles in this example) on the left, second type (boundary particle in this example) on the right.

<pre> int p=get_global_id(0); /* particle index */ for (int i=0; i<M; ++i) { /* index of the next fluid neighbor */ int neib_index=neibsList[i*N+p]; if (neib_index == NEIBS_END) break; /* do stuff with neib_index */ } </pre>	<pre> int p=get_global_id(0); /* particle index */ for (int i=M-1; i>=0; --i) { /* index of the next boundary neighbor */ int neib_index=neibsList[i*N+p]; if (neib_index == NEIBS_END) break; /* do stuff with neib_index */ } </pre>
---	---

4.4.2 Split kernels

Once traversal of individual particle types (for the system itself or even just for the neighbors) has been made efficient, the more complex kernels that need to provide significantly different behavior based should be split as well. For example, in GPUSPH we have recently refactored the main computational kernel (dedicated to the computation of the forces acting on each particle) into separate versions to compute fluid/fluid, fluid/boundary, boundary/fluid, etc. interactions. While this generally requires some additional memory access (because, e.g., the particle accelerations now need to be stored and retrieved between one incarnation of the kernel and the next), there has been an overall performance benefit; for the most complex formulations, on first-generation Kepler hardware, we have seen a 50% increase in the number of iterations per second. On the more recent and capable Maxwell architecture, the benefit has been less significant (30% more iterations per second), due to the smaller number of register spills: the more modern hardware supports more registers per work-item and is therefore less affected by the computational issues associated with particle system heterogeneity.

4.5 Multi-GPU

Very large particle systems can benefit from distribution across multiple GPUs. This may in fact be *necessary* simply due to the limited resources available on a single GPU: high-end GPUs currently have at most 16GB of RAM, which may limit the particle system size to a few tens of millions, depending on the complexity of the system. Even for smaller systems, however, distribution over multiple GPUs can provide a performance boost, provided each of the devices is saturated (otherwise, the overhead involved in distributing the particle system will be higher than the benefits offered by the higher computational capacity).

Distributing a particle system across multiple GPUs requires a significant change of vision: GPU coding is facilitated by its shared-memory architecture, where all compute units have read/write access to the entire device global memory. Multi-GPU introduces a distributed parallel computing layer, shifting the focus to efficient work distribution and data exchange between the devices.

For particle systems, the preferential way to distribute work across devices is based on domain decomposition rather than task decomposition, since the former allows both to minimize data exchange and to cover the associated latency more easily. Domain decomposition is achieved by distributing separate sections of the particle system to different devices (e.g., half of the particles and the associated data go on a GPU; the second half goes on a second GPU). When the particles can be processed independently (i.e., no neighborhood information is required), the partition is trivial, and the only objective is load balancing (i.e., ensuring that the fraction of particle system assigned to each GPU is proportional to its computational power). In these cases, the only data exchange needed between GPUs is the tracking of some global quantities, such as the particle system optimal position in PSO.

The decomposition becomes more challenging when the particles' behavior depends on a local neighborhood. In this case, each device must track the state not only of the particles that have been assigned to it but also their neighbors, some of which may have been assigned to other devices. Each device therefore has a view of the particle system as divided in four sections:

- (Strictly) inner particles: particles assigned to the device and that no other device is interested in; no information exchange is involved in their processing.

- Inner edge particles: particles assigned to the device that are neighbors of particles assigned to different devices; information about the evolution of these particles must be sent to other devices.
- Outer edge particles: particles assigned to other devices that are neighbors of particles assigned to this device; information about the evolution of these particles must be received from other devices.
- (Strictly) outer particles: particles assigned to other devices and that this device does not care about; no information exchange is involved in their processing.

The inner/outer relation is symmetrical, in the sense that a (strictly) inner particle for a device is (strictly) outer for all other devices and an inner edge particle for a device is an outer edge particle for at least one other device (**Figure 4**). We will say that two devices are adjacent if they share an inner/outer edge relation (i.e., if they need to exchange data about neighboring particles). Note that, depending on how the particle system is distributed, information about an inner edge particle may need to be sent to multiple adjacent devices.

4.5.1 Computing versus data exchange

The key to an efficient multi-GPU implementation is the ability to minimize the impact of data exchange. The most obvious approach in this sense is to minimize the data transfers themselves, for example, by ensuring that the domain is partitioned in such a way that the number of edge particles is minimized.

As a general rule, during the simulation each device will hold all the data relevant to both its inner (and inner edge) particles and the data relevant to its outer edge particles, i.e., the inner edge particles of adjacent devices. However, it will only run computational kernels on its own particles (using, read-only, the information from the outer edge particles) and then receive updates about the outer edge particles from the adjacent devices. On the other hand, there are cases when it may be convenient for each device to compute the updates for the outer edge particles by

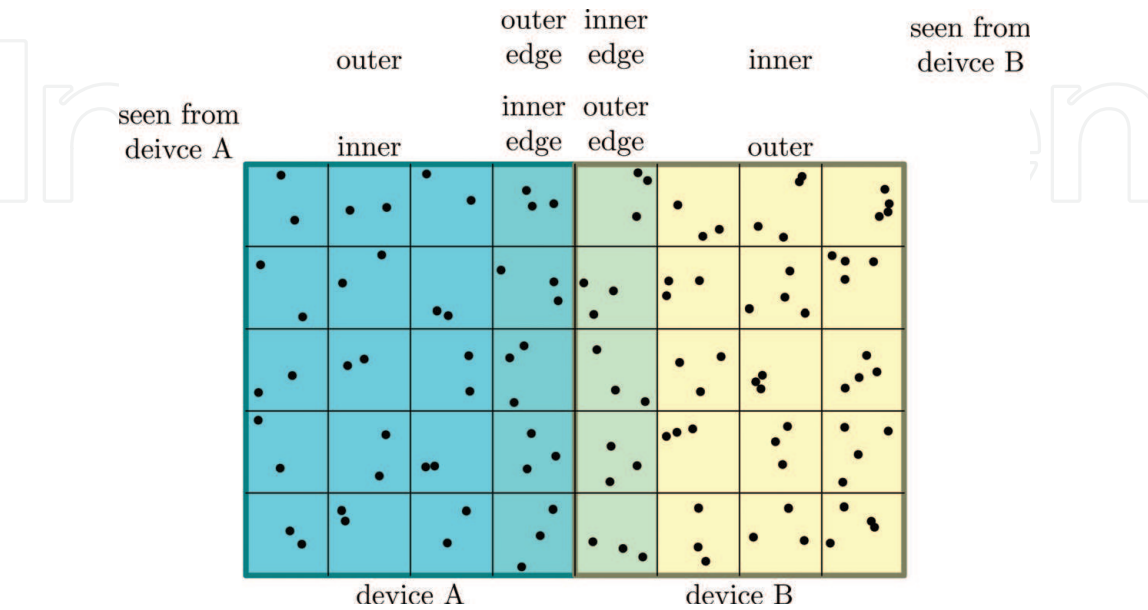


Figure 4. Inner/outer/edge relationship between devices, with domain decomposition based on a reference grid: inner/outer/edge particles are then defined based on the cell they belong to rather than on a purely geometrical relationship to the particles assigned to other devices.

itself. This can be done under the condition that the update does not require information from the neighbors (since the device does not hold information about all the neighbors of outer edge particles) and becomes convenient when the amount of data to transfer before the update is less than the amount of data to transfer after the update.

As an example, consider a simple particle system where the forces acting on each particle are computed from the interaction with the neighbors (forces kernel), but the new position and velocity are computed only from the previously computed forces (euler kernel), without any (further) interaction with the neighbors. Assume that there are two devices, D1 and D2, and that all the involved arrays (forces, positions, and velocities) are the same size (e.g., a float4 per particle). Then it is more convenient for D1 to get the information about the forces acting on its outer edge particles from D2 (and conversely), and then run euler on the outer edge particles, than it is for D1 to get the information about the positions and velocities after euler has been run on both devices: this is because exchanging forces results in a data transfer of a single float4 per (outer edge) particle, while exchanging positions and velocities would require exchanging twice as much data.

4.5.2 Computing during data exchange

Assuming data transfers have been minimized as discussed in the previous paragraph, the next step in an efficient multi-GPU implementation is to cover the data transfer latency by running computational kernels concurrently with the data transfer itself. This can be achieved as long as it is possible to efficiently launch computational kernels on a subset of the particle system (specifically, on the inner edge particles). It is then possible to first compute the new data on the inner edge particles, and then launch the kernel on the remaining (strictly inner) particles, while the inner edge data is sent to adjacent devices (and conversely the outer edge data is received from adjacent devices). This strategy allows optimal latency hiding, especially for the most computationally intensive kernels, provided all involved device are saturated. In GPUSPH, this is how we achieve nearly linear speedups in the number of devices [26], even when network transfers are involved in multi-node simulations [23].

4.5.3 Reference grid for domain decomposition

In our experience, multi-GPU also benefits from the use of the auxiliary grid that can be used for sorting (as described in Section 4.3) and for improved numerical accuracy (as will be described in Section 5.4): indeed, to improve the efficiency of data transfers, it is important that the sections of the arrays that need to be sent to adjacent devices are as consecutive as possible, since multiple small bursts are generally less efficient (both over the PCI Express bus and over the network) than larger data transfers.

With the auxiliary reference grid, this can be obtained by always splitting the domain at the cell level and computing the cell index by taking the inner/edge/outer relation into consideration (**Figure 4**). In GPUSPH this is achieved by reserving the two most significant bits of the cell index to indicate strictly inner cells (00), inner edge cells (01), outer edge cells (10), and strictly outer cells (11)—with the latter never actually used except in an optional global cell map. With this strategy, all strictly inner particles will be sorted first, followed by all inner edge particles, and finally by outer edge particles. Since outer edge particles will be sorted consecutively in memory, receiving data about them will be more efficient; similarly, sending inner edge data over will be made more efficient by the coalesced layout.

Note that this cell sorting strategy does not completely eliminate the need for multiple transfers; it does however help to reduce it significantly, especially when combined with linear cell indexing and the appropriate choice of order of dimensions for linearization. For this reason, GPUSPH offers a (compile-time) option to allow customization of this choice that in our experience can have performance benefits of up to 30%.

5. Numerical robustness

When the intent of GPGPU is to leverage the low-cost, high-performance ratio offered by consumer GPUs, a significant bottleneck in scientific applications is given by the limited (when not completely absent) support for double precision: since consumer GPUs are designed for video games and similar applications, where the highest rendering accuracy is not a requirement, the hardware is optimized for single-precision computation. Applications that need higher accuracy can thus follow one of the following strategies: use double-precision anyway, use soft extended precision (double-float, etc.), and rely on alternative algorithms that ensure a better utilization of the available precision.

Due to the high-performance cost (ratios as low as 1:32 compared to single precision, nearly completely defeating the benefits of the high performance of GPUs over CPUs), the use of double precision should be avoided whenever possible. If absolutely necessary, it should be restricted to parts of the code where it is essential. In all other cases, faster alternatives should be sought out. A possible solution is offered by double-float arithmetic, in which two single-precision values are used to provide higher accuracy [27, 28]: most operations will require between two and four hardware operation to complete, and the overall accuracy will generally be slightly lower than using actual double precision, but this can still be a good compromise between performance and accuracy when 64-bit floating-point has low or no support in hardware.

In many cases, it will be possible to avoid relying on extended precision by taking some care in the choice of algorithm used. In fact, the methods and algorithms that we will discuss momentarily can help improve the accuracy of an implementation regardless of the precision used: we would recommend their use even with double precision, since they always lead to more accurate results and in some cases (such as Horner's method) even higher performance.

5.1 Horner's method

Polynomial evaluation should always be done using Horner's method [29]. Any polynomial $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ can be written in the equivalent form

$$(((\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots)x + a_1)x + a_0.$$

When this second form is evaluated from the innermost to the outermost expression, better accuracy and performance can be achieved. Indeed, Horner's method is known to be optimal in that it requires the minimal number of additions and multiplications for the evaluation of the polynomial [30, 31], and given the widespread availability of fused multiply-add operations on modern hardware, every term can be computed with a higher accuracy and in a single cycle, making this evaluation method the fastest and most accurate for general polynomials.

5.2 Compensated and balanced summation

In many particle systems, the behavior of a particle is dictated by the influence of a local neighborhood: the total action on the central particle is then achieved by adding up the contributions from each neighboring particle. The number of contributions is frequently in the order of tens or hundreds and in some applications even more. The naive approach, which could be described algorithmically as

```
total_action=0;;
for (i=0; i<num_neighbors; ++i);
    total_action +=contribution_from_neighbor(i);
```

suffers from low accuracy: since each contribution adds a relative error in the order of the machine epsilon ϵ , the total relative error is in the order of the total number of contributions, $O(\epsilon n)$ in the worst case (in practice, the error is typically $O(\epsilon \sqrt{n})$ with the default round-to-nearest rounding mode).

This can be significantly improved by using a compensated summation algorithm, which can bring down the total relative error to order $O(1)$ (constant). The idea behind this class of algorithms is to keep track of the quantity that gets “lost” during a summation due to the finite precision. This is achieved by keeping two accumulators, the sum itself and a correction. The simplest form of compensated summation was popularized by Kahan [32], where the contribution from each new term is computed as shown in **Listing 5**.

The approach relies on the compiler not trying to do an algebraic simplification of the expressions for the temporary sum t and the correction, which may require disabling “fast-math” and the contraction of floating-point expressions.

The Kahan compensated summation algorithm works best when all terms in the summation are of similar or decreasing orders of magnitude but fail to take into account that the new term may be (significantly) larger in magnitude than the current running sum. To this end, Neumaier presented a variant [33], usually known as KBN (Kahan-Babuška-Neumaier), that takes into account the relative magnitude of the new term and the running sum when computing the correction (**Listing 6**). In contrast to Kahan’s algorithm, the final sum is obtained by adding sum and correction. More details about balancing and compensated summation algorithms can be found in [34].

The main downside of KBN is the branching condition to compute the correction, which may reduce performance on GPU. The algorithm can be rewritten to be vector-friendly, as illustrated on the right in **Listing 6**, which makes use of the `select(a, b, c)` function and the component-by-component extension to the ternary operator `c ? b : a` defined, e.g., in OpenCL C. This form of KBN should only be chosen when profiling shows the branching to be a performance bottleneck, since the extra operations otherwise introduce higher latency in the summation.

Listing 5.
Kahan summation algorithm.

<pre>y=term - correction; t=sum + y; correction=(t - sum) - y; sum=t;</pre>	<pre>/* take into account what got lost previously */ /* temporary sum: add the new term y */ /* estimate what got lost adding the new term */ /* actual new value of the summation */</pre>
---	--

Listing 6.
KBN (Kahan-Babuška-Neumaier) summation algorithm. Standard form (left) and possible vectorization (right).

<pre>t=sum + term; if (abs(sum)>= abs(term)) { correction +=(sum - t)+term; } else { correction +=(term - t)+sum; } sum=t;</pre>	<pre>t=sum + term; cond=(abs(sum)>= abs(term)); sum_or_term=select(term, sum, cond); term_or_sum=select(sum, term, cond); correction +=(sum_or_term - t)+term_or_sum; sum=t;</pre>
---	---

The downsides of compensated summation algorithms are higher storage requirements (the additional accumulator) and higher computational cost (Kahan, e.g., requires four times more operations compared to the standard summation). Compared to the use of double precision, the storage requirements are unchanged; the computational cost, however, is two (or more) times higher than the cost of double-precision on hardware that supports it at full frequency; on most consumer GPUs, however, the use of compensated summation can be up to eight times *faster* than the use of double precision.

Compensated summation algorithms can be used both locally, at the single kernel level, to improve the computation of the contributions for the next time step, and globally, across kernel launches, providing better accuracy for long-running simulations.

5.3 Vector norms and the hypot function

Computing the norm of a vector is a very frequent operation. In Euclidean metric, the norm is computed as the square root of the sum of the square of the components and thus requires d multiplications, $d-1$ additions, and a square root. With the exception of very high dimensions, the final square root is frequently the most expensive operation as well as the least accurate.

The first step to improve both performance and accuracy is therefore to avoid taking the square root if possible. For example, when the vector is a distance and the objective is to compare distances, it is much faster (and accurate) to compare the squared distances (sum of the squares of the differences of the components) rather than the distances themselves. When the distance has to be compared against a reference length, it is cheaper to square the reference length than it is to take the square root in the distance computations.

A typical circumstance where one cannot avoid taking the square root is normalization of a vector, in which each component needs to be divided by the length of the vector; in some applications this is such a frequent (and slow) operation that fast, but less accurate implementations are used, such as the fast inverse square root [35] popularized by its use in id Software *Quake III: Arena* video game [36].

While games can afford to sacrifice accuracy for performance, this is not the case in scientific applications, for which a significant issue in vector normalization (and similar operations) is numerical stability: when the vector has components which are very close to zero, a naive computation of the norm may lead to underflow, potentially resulting in a final division by zero during normalization; conversely, very large components can lead to overflow of the inner summation before the root extraction.

The solution is to compute the norm using the hypot operator. The idea is to rewrite $\sqrt{a_0^2 + a_1^2 + \dots + a_n^2}$ as $|a_0| \sqrt{1 + q_1^2 + \dots + q_n^2}$ where $q_i = a_i/a_0$. In exact arithmetic the two expressions are equivalent, but with finite precision, the second expression is more accurate, assuming the values are sorted by magnitude (largest to smallest). The higher accuracy of hypot however comes at a significant computational cost, due to the additional division per component: even highly optimized implementations of hypot (such as the two-argument function available in the standard C library, in OpenCL C and in CUDA) can easily be as much as two orders of magnitude slower than the naive approach.

5.4 Local versus global position

A major difference between numerical methods such as finite differences and meshless methods such as smoothed particle hydrodynamics is that in the former case, there is no need to track the global position of each computational node, as this is defined algorithmically based on the (possibly adaptive) mesh size, and the internode distance is fixed and known in advance. Meshless methods, on the other hand, need to track the global position of each particle; due to the nonuniform distribution of floating-point values, the inter-particle distance (computed as the difference between the global position of the particles) will then have a higher precision near the origin of the domain and a lower precision the further away from the origin the particles are. When the ratio of the inter-particle distance to the domain size gets close to machine epsilon, this nonuniform accuracy may lead to artificial clustering of particles, an effect that is quite noticeable when using single precision to simulate very large domains with a very fine resolution (**Figure 5**).

While extending the precision for the global position is a possible solution [37], this only delays the problem and, as mentioned previously, may have a nontrivial computational cost. An alternative approach is to use a support, fixed grid with regular spacing and only track the position of each particle with respect to the center of the grid cell it belongs to [38]; distances to the other particles are obtained by correcting the distance between the grid cell centers and the local position difference (**Listing 7**). Absolute (global) positions are only reconstructed when necessary (e.g., when writing the results to disk).

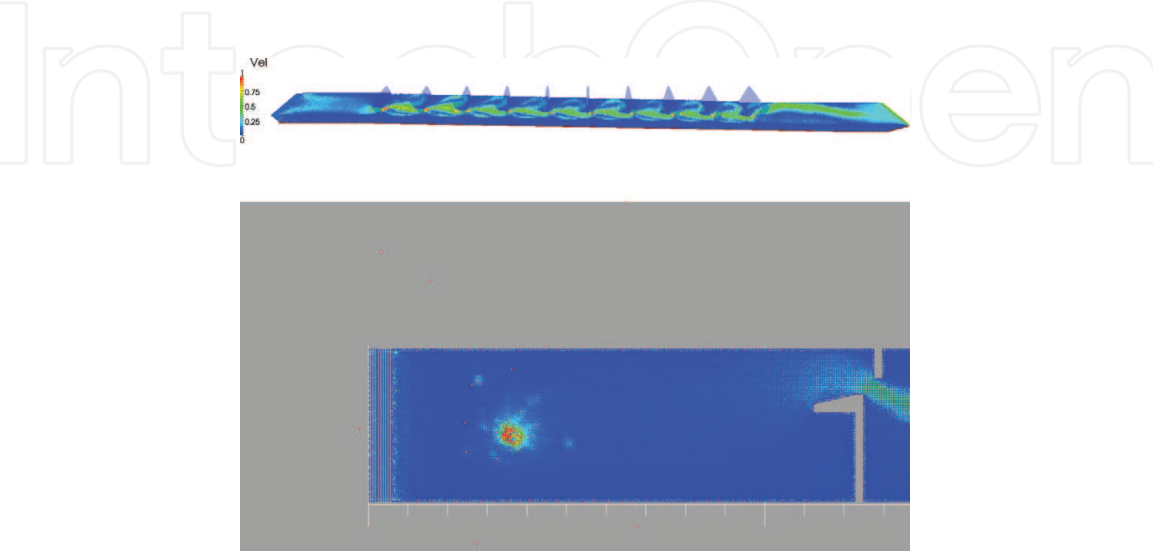


Figure 5. Simulation of an 8-pool fish pass with GPUSPH before the introduction of uniform position precision. At high resolutions, spurious explosion due to insufficient precision near the domain edge can be observed.

Listing 7.

Computing particle distance with uniform precision using cell indices and local positions.

```

const float3 cell_size;           /* global constant: cell size */
int3 our_cell, neib_cell;         /* (integer) coordinates of the cells */
float3 our_lpos, neib_lpos;       /* particle position wrt their cell center */
float3 dist_vec =                 /* particle distance, computed from the local
    (neib_cell - our_cell)*cell_size +    position difference and the cell center
    (neib_lpos - our_lpos);              distance */

```

This approach doubles the storage requirement (e.g., in three dimensions, an additional `int3` is needed to describe the cell index, in addition to the local position stored in a `float3`), matching the requirement of the higher-precision type (e.g., storing global positions but using a `double3` per particle), with the additional benefit of uniform accuracy throughout the domain, and the possibility to support much larger domains, even larger than would be allowed with higher precision. If the overall number of cells is relatively low, storage requirements can be reduced by encoding the cell coordinates in less memory: for example, if the overall number of cells is less than 2^{32} , it is possible to store a linearized cell index in a single unsigned `int`. The support grid solution is particularly advantageous when the same grid (and linearized cell index) can also be used for neighbor search, as described in Section 4.3, and to improve the performance of multi-GPU simulations, as discussed in Section 4.5.

5.5 Relative and nondimensional quantities

A general rule to improve the numerical stability of most methods is to work in nondimensional form, i.e., to scale all quantities (length, time, mass, etc.) by some problem-specific scale factor related to the problem's characteristic numbers (e.g., the Reynold number in case of viscous flows), and rewrite the underlying equations in terms of the nondimensional quantities instead of using standard units.

Working with nondimensional quantities can lead to better accuracy by allowing fuller utilization of the range of the floating-point values, but additional steps can be taken to gain further accuracy. An example of this is the local coordinate system proposed in the previous paragraph that alone is sufficient to improve the energy conservation of GPUSPH by as much as four orders of magnitude [38], but the same principle can be extended to most quantities. The benefits are particularly high when the range of variation of the quantity itself is small.

For example, in the weakly compressible SPH formulation, the density ρ of the particles in a fluid is assumed to differ from the reference (at-rest) value ρ_0 by at most a few percent. In nondimensional terms, the recommended approach would be to work with the *relative density* $RD = \rho/\rho_0$, but the numerical accuracy can be further improved by working with the *centered relative density* $\tilde{\rho} = \rho/\rho_0 - 1$, which allows us to gain three or more digits of accuracy and to bring the absolute error in the computation of the neighbor contribution to the pressure part of the momentum equation from 10^{-7} down to 10^{-11} [38].

6. Flexibility

When setting forth to implement a new particle engine, important decisions have to be made concerning the general design of the system, particularly in reference to the scope and objectives. This is particularly important for particle system,

due to the possible temptation to produce a “universal particle system engine” that could be extended to support any kind of particle system: a worthy objective, but in direct contrast with the need to have something useful and functional “right now.” As a general rule, our recommendation is to start with a well-focused objective (e.g., a very specific formulation of a numerical method) and only extend/expand the objective for specific use cases.

The focus of the initial implementation of a particle system (as for any general application) should always be correctness over performance. This is true also when specifically targeting high-performance computing (e.g., when designing for a GPU implementation). There are however some important design aspects that can be kept in mind, with low implementation cost and high return of investment at later development stages, both in terms of code cleanliness and extensibility. We will present them in this section, showing how the complexity that comes with flexibility can be isolated to provide a cleaner interface and without sacrificing (runtime) performance.

6.1 Separation of roles

Implementations of particle systems can be characterized by three main roles: *management* (setup, teardown, data exchange, e.g., saving/visualization), *evolution* (abstract sequence of steps that describe a single iteration of the lifetime of the system), and *execution* (concrete functions and kernels for each individual step). Keeping these roles distinct from the beginning can provide a solid base for the growth of the implementation; for example, while at first the developer may focus on single GPU, a subsequent extension to multi-GPU can be achieved more easily at a later stage if the management and execution roles are delegated to separate classes, running on separate threads: typically, management is handled by the main thread, while the execution role will be delegated to a worker thread (which become multiple worker threads in the multi-GPU case).

Two approaches are then possible: assign the evolution role (i.e., the decisions about which steps to run next) to the manager thread or to the workers. The latter choice (“smart workers”) has the advantage that the worker threads know what to do for every step, and this can be leveraged to reduce synchronization points; the downside is that unification tasks (such as global reductions and data exchange in the multi-node case) must be taken over by a specific worker, which creates an imbalance (since not all workers are created equal). Conversely, with “dumb workers,” most commands become a synchronization point (which can become a performance bottleneck), but the manager thread can more easily subsume the unification tasks. Hybrid solutions are also possible, at the expense of a growing complexity in logic. Factoring out the (abstract description of the) evolution into its own class (in the object-oriented programming sense) can make it easier to transition from one implementation to the other.

6.1.1 Workers for hardware abstraction

Refactoring the execution role into a separate Worker class has several advantages. The most important, as mentioned above, is that having a separate Worker class provides a solid ground to expand the code to support multiple GPUs (with each Worker taking control of a separate device).

Additionally, with the correct design, wider hardware support can be implemented by making the Worker class itself an abstract class, implementing only the common code, such as the evolution logic in the case of smart workers, or the command dispatch table in the case of dumb workers. Derived classes would then

implement the hardware-specific details such as the actual kernel execution or the host/device memory transfers. For example, one could have a GPUWorker for execution on GPU and a CPUWorker for execution on CPU [26]; the GPUWorker class itself could be further specialized in a CUDAWorker and an OpenCLWorker, when support for both APIs is desired.

6.2 The cost of variation

The complexity of the implementation of each individual step of the evolution loop in the particle system is directly related to the number of features offered; for example, some sections of the code may only be relevant when the fluid needs to interact with moving objects; or some particle types or particle data may only be present if specific options (e.g., an SPH formulation) are enabled; or it may be possible to choose between a faster but less accurate approach and a computationally more expensive, more accurate solution.

There are two main costs that come with providing multiple features (or variations thereof): an implementation cost (larger, more complex code to write) and a runtime cost. Both costs can be reduced with appropriate care in the design of the software. An optimal implementation should be designed in such a way that the runtime cost of a disabled feature matches the runtime cost had it not been implemented in the first place. For example, if the user runs an SPH simulation without any moving objects, then the runtime should be the same in an SPH implementation that does not support moving objects, and in an implementation that does support them, but that can detect (or can be told) that support for them is not needed. Likewise, the implementation of any additional feature should be as unobtrusive as possible, factored out into its own sets of functions. Luckily these two goals are not in conflict.

6.2.1 Managing buffers

One of the key aspects in supporting multiple variants for particle systems is correct buffer management. The objective is to support allocating all and only the (copies of the) buffers that are needed, correctly sized, in a unified manner (i.e., with the same interface on host and device). Moreover, we want to be able to pass around sets of buffers (e.g., the collection of all allocated buffers or a specific subset of them) to other parts of the code, in a way that minimizes both the actual copying of data and the detailed specification of which buffers belong to a set.

The approach we use in GPUSPH to solve this issue relies on two aspects: bitmask-based buffer naming and a set of classes that abstract buffer management, isolating the details of the individual buffer while still allowing the retrieval of all the necessary information, such as the data type, the number of elements, the kind of buffer (e.g., host or device), etc.

Bitmask-based buffer naming relies on using an appropriate set of values (defined with either `#defines` or an `enum`) to refer to the buffers, on the condition that each (symbolic) buffer name corresponds to an individual bit. These symbolic buffer names are used to refer to buffers in most of the host code, with the exceptions of the classes and functions that require access to the actual corresponding data pointers (i.e., the lowest level of implementation of the GPUWorker). An example of this is shown in **Listing 8**, which needs to be paired with an appropriate BufferList class such that, given BufferList buffers, we can get the array of positions as `buffers.getData<BUFFER_POS>()`.

With each symbolic name associated to a separate buffer, it is possible to combine them into an expression to refer to multiple buffers at once. For example,

Listing 8.
Buffer as member variables (left) versus buffer symbolic names (right).

<code>float4 *bufferPos;</code>	<code>#define BUFFER_POS (1U < 0)</code>
<code>int *bufferNeibs;</code>	<code>#define BUFFER_NEIBS (1U < 1)</code>
<code>float2 *bufferTau[3]; /* stress tensor */</code>	<code>#define BUFFER_TAU (1U < 2)</code>

BUFFER_POS | BUFFER_VEL would indicate a collection of buffers holding both the positions and the velocity. This is particularly useful in conjunction with the “dumb worker” approach, because it allows most commands given by the Manager thread to the Workers to have a syntax such as

```
doCommand(COMMAND_NAME, BUFFER_R1 | BUFFER_R2 | ...,
          BUFFER_W1 | BUFFER_W2 | ...);
```

where the list of input and output buffers for the command are given as single parameters by doing a binary OR of the symbolic buffer names.

In languages such as C++, the use of symbolic names (with or without the bitmask property) also allows static knowledge about the buffer properties to be encoded into “traits” structure that can be used to evince information about the buffers at compile time, as exemplified in **Listing 9**. This allows developers to programmatically determine the element type of a buffer as `BufferTraits<symbolic_name>::element_type`, which can be used in our `BufferList` class to make sure that when requesting a specific array, we get a pointer of the correct type and `BufferTraits<symbolic_name>::num_buffers` to determine how many components the buffer has (e.g., in this example we model a symmetric 3×3 tensor, which has six components, as a collection of three `float2` buffers).

The management of the actual data is handled with different layers of abstraction. We use an `AbstractBuffer` class to describe the interface shared by all buffers, regardless of content or type; the interface presents (pure virtual) methods for operations such as allocations and deallocation of data, as well as a way to return a “generic” pointer to the data itself (as a `void*`, since no type information is available in `AbstractBuffer`).

The next layer can be implemented as a `GenericBuffer` class template that depends on the data type and number of components of the buffer, so that `GenericBuffer<float2, 3>` would be able to handle storage and typed access to per-particle symmetric tensor data (encoded in three `float2` per particle).

Listing 9.
Example declaration of a BufferTraits structure and its specializations for some named buffers, declaring their data type and multiplicity.

<code>template<int Buffer> struct BufferTraits;</code>	<code>template<> struct BufferTraits<BUFFER_NEIBS></code>
	<code>{</code>
	<code> typedef int element_type;</code>
	<code> enum {num_buffers=1};</code>
	<code>};</code>
<code>template<> struct BufferTraits<BUFFER_POS></code>	<code>template<> struct BufferTraits<BUFFER_TAU></code>
<code>{</code>	<code>{</code>
<code> typedef float4 element_type;</code>	<code> typedef float2 element_type;</code>
<code> enum {num_buffers=1};</code>	<code> enum {num_buffers=3};</code>
<code>};</code>	<code>};</code>

Since we can derive element types and number of components from the buffer traits, our Buffer class template can simply derive from the appropriate GenericBuffer:

```
template<flag_t Key>
class Buffer : public GenericBuffer<
    BufferTraits<Key>::element_type,
    BufferTraits<Key>::num_buffers>
{ /* other specializations, as necessary */ };
```

Note that the Buffer class template still does not have any actual allocation logic: its only purpose is to provide the correct base class for the named properties of each particle (e.g., position, velocity, etc.). The allocation logic is delegated to derived classes such as HostBuffer (that would use malloc/free or new/delete), CUDABuffer (using cudaMalloc/cudaFree), and CLBuffer (using clCreateBuffer/clReleaseMemObject).

Finally, a collection of buffers is managed by a BufferList that maps symbolic names to the concrete class implementing the buffer with the specific symbolic name.

Since different variants of a particle system will instance different subsets of all possible buffers, the mapping will in general be sparse, and it is therefore better to use a dictionary type such as std::map (or language equivalent) to implement it. This is particularly true for the bitmask choice of symbolic names. Moreover, since each symbolic name maps to a buffer with a different data type, the mapping will generally be between symbolic names and AbstractBuffers. Downcasting to the correct Buffer<symbolic_name> type can be done in specific template methods, where the symbolic name is known as compile type. Our BufferList class, for example, exposes a template getData method that returns a pointer of the correct type for the given symbolic name, again using the buffer traits to deduce it; some example of its usage are illustrated further on.

In general, a single instance of a running particle system will have multiple BufferLists: it may have one to hold the data on host (e.g., for saving or visualization) and one on device to hold the data for the running simulation; on device, it might have more than one, separating read/write copies of each buffer, or to hold the data for different parts of the particle system (e.g., a BufferList for fluid particle data, a BufferList for boundary particle data); it may also have one (or more) BufferLists to hold all of the available data for the particle system and separate smaller BufferLists with a selection of the data when passing it to individual kernels, depending on the approach used.

6.2.2 Handling options combinatorial growth

As the number of options offered to the end user of a particle system grows, there is a consequent explosion in the number of possible valid combinations that need to be supported, which results in competing needs between the opportunity (for performance reasons) for their compile-time implementation, and the associated resource consumption at compile time when runtime selection of the specific option is offered.

(At some point of time, compiling all of the combinations of simulation parameters offered by GPUSPH took several hours, occupying several gigabytes of memory and producing a binary with around 3000 variants of the main computational kernels overall.)

There are two possible solutions to this issue. The simplest solution, which is currently used in GPUSPH, is to push down the compile-time selection to the user: the setup of the user simulation is done via a source file where all the compile-time parameters must be selected. When the user simulation setup is compiled, only the specific combination of parameters will be enabled and compiled for. An alternative solution is to rely on the specific possibility offered with GPU programming, to compile the device code at runtime. This feature has always been available with OpenCL (in fact, OpenCL *requires* runtime compilation of the device code), and it has been recently made available in CUDA via the NVRTC library.

The main downside of the compile-time selection is that the software must always be distributed in source form, with several implications in terms of user-friendliness and maintenance. Downsides for runtime compilation include the limited support for older version of CUDA, when relying on this proprietary solution, and the potential time loss at the beginning of each execution (which may or may not be offset by any caching the runtime compilation engine might do).

On the other hand, runtime compilation of the device code allows an even wider range of aspects to be implemented at compile time (on the device side), which may allow even stronger compiler optimizations. For example, global simulation parameters that are constant throughout a simulation are often stored in the device constant memory at the beginning of the simulation; even though access to constant memory is quite efficient, inlining the constants can be even more efficient, and this can be achieved exploiting runtime compilation, by replacing the upload of the constants to the device with appropriate `#define` in the runtime-compiled device code.

6.2.3 C++ SFINAE versus C preprocessors for compile-time specialization

Implementing multiple variations of a kernel (or function used by a kernel) is generally nontrivial, as the functions may have different sets of parameters and different private variables and may operate differently even on data that is present in all or most of them. The objective is therefore to isolate the variant-specific parts from the common parts, avoiding code repetition.

When using runtime compilation for the device code and a C-based language such as OpenCL C, the only way to achieve this is to fence nonrelevant parts of the code with appropriate preprocessor directives, as illustrated in **Listing 10** (left). Code fencing can be factored out, and sometimes reduced, by collecting data into conditional structures and refactoring computations into conditional functions; the resulting code is slightly more verbose (**Listing 10**, right), but the optional features are better isolated, improving the maintainability of the code.

Note that the conditional structure in this example cannot be extended to the kernel parameters and private variables itself, due to the impossibility for global array addresses to be member of structures shared between host and device, which further limits the possibility to initialize the conditional parts of the private variable structure with appropriate conditional functions. (This is a limitation of OpenCL C; alternative solutions are possible using the Shared Virtual Memory feature introduced in OpenCL 2.0 and supported by some implementations as an extension on older versions of the standard.)

When the device code can be written in C++ and global arrays pointers are made available in the same form to both the host and device (e.g., with CUDA), the language itself provides powerful meta-programming techniques that can be leveraged to eliminate the need for a preprocessor, allowing multiple specialized implementations to coexist.

Listing 10.

Code fencing for optional components in the case of runtime device code compilation: inline approach (left) and refactored approach with code isolation (right).

<pre> kernel void some_kernel(global const float4 * restrict posArray, global const float4 * restrict velArray, #ifdef HAS_XSPH global float4 * restrict xsphArray #endif #ifdef HAS_STRESS_TENSOR global float4 * restrict stressTensor4, global float4 * restrict stressTensor2, #endif global float4 * restrict forces) { /* common private variables go here */ #ifdef HAS_XSPH /* XSPH private variables go here */ #endif #ifdef HAS_STRESS_TENSOR /* stress tensor private variables go here */ #endif /* common computations go here */ #ifdef HAS_XSPH /* XSPH computations go here */ #endif #ifdef HAS_STRESS_TENSOR /* stress tensor computations go here */ #endif } </pre>	<pre> struct private_vars { /* common private variables */ #ifdef HAS_XSPH /* XSPH private variables */ #endif #ifdef HAS_STRESS_TENSOR /* stress tensor private variables */ #endif }; void process_common(struct private_vars *priv) { /* common computations here */ } void process_xsph(struct private_vars *priv) #ifdef HAS_XSPH { /* XSPH computations go here */ } #else {} /* intentionally left blank */ #endif void process_stress_tensor(struct private_vars *priv) #ifdef HAS_STRESS_TENSOR { /* stress tensor computations go here */ } #else {} /* intentionally left blank */ #endif kernel void some_kernel(global const float4 * restrict posArray, global const float4 * restrict velArray, #ifdef HAS_XSPH global float4 * restrict xsphArray #endif #ifdef HAS_STRESS_TENSOR global float4 * restrict stressTensor4, global float4 * restrict stressTensor2, #endif global float4 * restrict forces) { struct private_vars priv; /* initialize common part of priv */ #ifdef HAS_XSPH /* initialize XSPH part of priv */ #endif #ifdef HAS_STRESS_TENSOR /* initialize stress tensor part of priv */ #endif process_common(&priv); process_xsph(&priv); process_stress_tensor(&priv); } </pre>
---	---

Conditional structures and functions in C++ can be implemented by using templates and the meta-programming feature of the language known as SFINAE (substitution failure is not an error) to select function specializations based on any combination of (compile-time) properties of their parameters. The approach we show requires two building blocks that are available in the standard library since C++11 (and can even be implemented in older C++ versions) and a special empty structure template.

The first building block is a structure template.

```
template<bool B, typename T, typename F>struct conditional;
```

such that `conditional<somecondition, SomeType, SomeOtherType>::type` corresponds to `SomeType` when `somecondition` is true and to `SomeOtherType` when the condition is false. This is part of C++11 and can be also defined in previous versions of the standard [39].

The purpose of the empty structure template is to “absorb” any type, construct from anything, and otherwise be empty. Using C++11 variadic templates for the constructor, it can be implemented as.

```
template<typename T>struct empty {
    template<typename U...>empty(U... args) {}
};
```

In older versions of C++, the single variadic template constructor must to be replaced with multiple constructor templates, each taking a separate number of arguments.

With these building blocks, we can define our conditional structure support type, relying on the C++11 using template directive:

```
template<bool B, typename T>
using cond_struct=typename
    conditional< B, T, empty<T> >::type;
```

When forced to use older C++ versions, something similar but less robust can be implemented with a macro such as.

```
#define COND_STRUCT(cond, ...) \
    typename conditional<cond, __VA_ARGS__, \
        empty<__VA_ARGS> >::type
```

A structure with optional members can then be defined by defining multiple structures grouping each set of optional members and then defining the main structure as derived from all substructures, each wrapped in their own `cond_struct<>`, as illustrated in **Listing 11**. We see how the individual groups of members for the final structure are refactored into simpler structures, carrying their own initialization information. We also see why the empty structure needs to be a template: if this was not the case, and both XSPH and the stress tensor computation were disabled, the kernel parameters structure would have `empty` as a base class twice, which is not allowed by the standard; with our template approach, the two empty base classes are now formally distinct types: `empty<xsph_kernel_params>` and `empty<stress_kernel_params>`. The sample code also shows the advantage of the `BufferList` class described previously and its typed buffer access methods. On the device side, we can use the same approach for the private variables of the kernel (**Listing 12**).

Finally, we need to define the individual process functions. For this, we need separate overloads depending on whether the `priv` structure has the specific members or not. One way to achieve this is to make all functions depend on the same template parameters as the structure, but when there are many parameters, this becomes quite complex and hard to maintain and extend, since every additional parameter will require a change in all the functions that access

Listing 11.

Conditional structures with C++ applied to kernel parameters: definition of the optional members (left) and definition of the conditional structure template including them (right).

```

struct common_kernel_params {
    const float4 * restrict posArray;
    const float4 * restrict velArray;
    float4 * restrict forcesArray;

    common_kernel_params(BufferList& buffers)
: posArray(buffers.getData<BUFFER_POS>())
, velArray(buffers.getData<BUFFER_VEL>())
, forcesArray(buffers.getData<BUFFER_FORCES>())
    {}
};

struct xsph_kernel_params {
    float4 * restrict xsphArray;

    xsph_kernel_params(BufferList& buffers)
: xsphArray(buffers.getData<BUFFER_XSPH>())
    {}
};

struct stress_kernel_params {
    float4 * restrict stressTensor4,
    float4 * restrict stressTensor2,
    stress_kernel_params(BufferList& buffers)
: stressTensor4(buffers.getData<BUFFER_TAU4>())
, stressTensor2(buffers.getData<BUFFER_TAU2>())
    {}
};

template<
    /* actual template parameters */
    bool needs_xsph, bool needs_stress_tensor,
    /* pseudo-template parameters,
    used to give simpler names to
    conditional structure members */
    typename optional_xsph=
cond_struct<needs_xsph, xsph_kernel_params>,
    typename optional_stress =
cond_struct<needs_stress_tensor,
    stress_kernel_params>
>
struct kernel_params
: common_kernel_params
, optional_xsph
, optional_stress
{
    /* These static variables allow compile-time
    knowledge about the parameters used
    for the specific instantiation
    of the template */
    static const bool has_xsph=needs_xsph;
    static const bool has_stress=
needs_stress_tensor;

    kernel_params(BufferList& buffers)
: common_kernel_params(buffers)
, optional_xsph(buffers)
, optional_stress(buffers)
    {}
};

```

the structure, regardless of whether the additional parameter actually has an impact.

A simple way is to make the functions into templates depending on a single parameter (the arbitrary type of the structure passed), and then make overloads based on specific properties of the actual structure that gets passed. This can be achieved by means of `enable_if`, a structure template declared as

```
template<bool B, typename T=void> enable_if;
```

which is such that `enable_if<condition, SomeType>::type` is `SomeType` when the condition is true and an error otherwise. Due to the SFINAE principle, when the compiler is looking for the overload of a function to use, it will discard (without errors) the overloads which result in an error and automatically select the one which does not result in an error. Additionally, if `SomeType` is omitted, `void` is implied, which can simplify the syntax. Again, this template is provided by the standard library in C++11 and can be implemented in older version of C++ [40].

To further simplify the syntax, we assume that C++11 is available and we can define:

```

template<bool B, typename T=void>
using enable_if_t=typename enable_if<B, T>::type;
(which is pre-defined in C++14).

```

Listing 12.
Conditional structures with C++ applied to private function variables: definitions of the optional members (left) and definition of the conditional structure template including them (right).

<pre>struct common_kernel_priv { /* common variables become members of this structure */ common_kernel_priv(common_kernel_params const& params) /* initialize the variables from the parameters */ }; struct xsph_kernel_priv { /* XSPH-specific variables become members of this structure */ xsph_kernel_priv(x sph_kernel_params const& params) /* typically, feature-specific variables will be initialized from feature-specific kernel parameters */ }; struct stress_kernel_priv { /* Stress-tensor specific variables become members of this structure */ stress_kernel_priv(stress_kernel_params const& params) /* typically, feature-specific variables will be initialized from feature-specific kernel parameters */ };</pre>	<pre>template< bool needs_xsph, bool needs_stress_tensor, typename optional_xsph = cond_struct<needs_xsph, xsph_kernel_priv>, typename optional_stress = cond_struct<needs_stress_tensor, stress_kernel_priv> > struct kernel_priv : common_kernel_priv , optional_xsph , optional_stress { /* These static variables allow compile-time knowledge about the parameters used for the specific instantiation of the template */ static const bool has_xsph=needs_xsph; static const bool has_stress= needs_stress_tensor; kernel_priv(kernel_params<needs_xsph, needs_stress_tensor> const& params) : common_kernel_priv(params) , optional_xsph(params) , optional_stress(params) {} };</pre>
---	--

The processing functions can then be defined as in **Listing 13**, with separate overloads made available based on compile-time properties of the argument. The kernel structure becomes very simple (**Listing 14**): all of the complexity has been delegated to specific (sub)structures and functions, and we have a guarantee that each specialized version of the kernel will only contain the code and variables that are pertinent to its functionality.

Listing 13.
Function specialization with overloads based on argument properties with enable-if in CUDA.

<pre>/* process_xsph is defined differently, depending on whether XSPH is enabled or not; we check for this based on the static const has_xsph member of the priv structure: this will always be present, and it will be true or false depending on whether XSPH was enabled */ template<typename Priv> __device__ enable_if_t<Priv::has_xsph> void process_xsph(Priv& priv) /* XSPH computations here */ template<typename Priv> __device__ enable_if_t<not Priv::has_xsph> void process_xsph(Priv& priv) /* intentionally left blank */</pre>	<pre>/* Similarly for the stress tensor, using has_stress */ template<typename Priv> __device__ enable_if_t<Priv::has_stress> void process_stress_tensor(Priv& priv) /* stress tensor computations here */ template<typename Priv> __device__ enable_if_t<not Priv::has_stress> void process_stress_tensor(Priv& priv) /* intentionally left blank */ /* The common code needs no special treatment */ __device__ void process_common(common_priv& priv) /* common computations here */</pre>
--	---

Listing 14.*Kernel structure after isolation of the optional components.*

```

template<
    bool needs_xsph, bool needs_stress,          /* template parameters for the kernel */
    typename Params =                           /* shorthand for the kernel parameters struct */
    kernel_params<needs_xsph, needs_stress_tensor>,
    typename Priv =                             /* shorthand for the kernel private variables */
    kernel_priv<needs_xsph, needs_stress_tensor>
>
__global__ void some_kernel(Params params)      /* kernel signature */
{
    /* initialize both common and optional private
    variables here */
    Priv priv(params);
    /* run common and optional parts of the code */
    process_common(priv);
    process_xsph(priv);
    process_stress_tensor(priv);
}

```

Listing 15.*Runtime switch to call the appropriate compile-time kernel specialization.*

```

if (opt_xsph && opt_stress) some_kernel<<<...>>>(kernel_params<true, true>(buffers));
else if (opt_xsph && !opt_stress) some_kernel<<<...>>>(kernel_params<true, false>(buffers));
else if (!opt_xsph && opt_stress) some_kernel<<<...>>>(kernel_params<false, true>(buffers));
else if (!opt_xsph && !opt_stress) some_kernel<<<...>>>(kernel_params<false, false>(buffers));

```

Runtime selection of the variant of the kernel to be used can be achieved with simple conditionals (**Listing 15**). However, when the number of conditionals is large, this can be rather bothersome to write; more compact and efficient solutions to the runtime switch are possible, using the meta-programming techniques presented in [41].

When using C macros, the multiple specialized variants of the kernel and related structures and functions cannot coexist in the same compilation unit (since C does not support overloading or templates), making runtime selection of the compile-time variant impossible: a single specific instance must be selected when the device code is compiled; on the upside, one would generally use C when using OpenCL C, for which the device code is compiled at application runtime, as discussed in the previous section.

In terms of syntax, the only significant downside of the SFINAE approach is that the signature needs to be repeated for every specialization, in contrast to the C macro approach, for which we only need one signature, and each implementation is fenced by `#if/#elif/#else/#endif`. This could be avoided using the C++17 feature `if constexpr`, but support for it in device code is still missing.

7. Conclusions

Particle systems are a fundamental aspect of many applications and numerical methods. By their own nature, they benefit from the massively parallel stream processing architecture of modern GPUs, but naive implementations can easily encounter pitfalls that can limit the full exploitation of the hardware.

While hardware vendors go to great lengths to support more liberal coding, the software can—and should—be designed to leverage the natural programming model of the hardware, and we have provided several insights on how the particle systems can be designed to fit better with the requirements of optimal GPU usage. We also presented a few simple ideas that, when taken into consideration during an initial implementation, can make future extensions much easier. Many of the suggestions we provide can also be of general interest beyond the implementation of particle systems.

We have stressed the importance of the choice to the correct approach in dealing with the potentially severe limitations in the numerical robustness of the implementation, due to the restricted accuracy and precision of the single-precision floating-point format which GPUs are optimized for. While many of the techniques we have presented are not new (some going as far back as the nineteenth century), in our experience they have surprisingly limited adoption; we hope that our discussion of their usefulness in this context will lead to higher awareness of the possibilities they offer. We dislike the adoption of higher-precision data types as a solution to the issue, not only because of the performance implications on consumer hardware, but as a philosophical objection to waste: why use the wrong numerical approach, *wasting* the additional precision granted by double precision, when the correct approach can make single precision sufficient? We do understand the need for the extended precision requirements in many applications, and we are sure that our reminiscence of classical solutions to better accuracy can benefit them as well, particularly since support for even higher-precision data types is nearly nonexistent in hardware (with the possible exception of the IBM POWER9 support for IEEE-754-compliant 128-bit floating-point formats) and especially on GPUs.

Author details


Giuseppe Bilotta^{1*}, Vito Zago¹ and Alexis Hérault²

¹ Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania, Italy

² Conservatoire National des Arts et Métiers, Laboratoire Modélisation mathématique et numérique, Paris, France

*Address all correspondence to: giuseppe.bilotta@ingv.it

IntechOpen

© 2018 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. 

References

- [1] Reeves WT. Particle systems—A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*. 1983;2(2):91-108. DOI: 10.1145/357318.357320
- [2] Paramount. *Star Trek II: The Wrath of Khan* [film]. 1982
- [3] Unity Technologies. Particle Systems. In: *Unity User Manual v2018.2* [Internet]. 2018. Available from: <https://docs.unity3d.com/Manual/index.html> [Accessed: July 18, 2018]
- [4] Monaghan JJ. Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*. 2012;44:323-346. DOI: 10.1146/annurev-fluid-120710-101220
- [5] Chen JS, Liu WK, Hillman MC, Chi SW, Lian Y, Bessa MA. Reproducing kernel particle method for solving partial differential equations. In: Stein E, Borst R, Hughes TK, editors. *Encyclopedia of Computational Mechanics*. 2nd ed. Chichester, UK: Wiley; 2017. DOI: 10.1002/9781119176817.ecm2104
- [6] Tiwari S, Kuhnert J. Finite pointset method based on the projection method for simulations of the incompressible Navier-Stokes equations. In: Griebel M, Schweitzer MA, editors. *Meshfree Methods for Partial Differential Equations*. Lecture Notes in Computational Science and Engineering. Vol. 26. Berlin, Heidelberg: Springer; 2003
- [7] Bićanić N. Discrete element methods. In: Stein E, Borst R, Hughes TK, editors. *Encyclopedia of Computational Mechanics*. Chichester, UK: Wiley; 2017. DOI: 10.1002/0470091355.ecm006.pub2
- [8] Kennedy J, Eberhart RC. Particle swarm optimization. In: *Proceedings of ICNN'95—International Conference on Neural Networks*; November 27– December 1, 1995; 2002. pp. 1942-1948. DOI: 10.1109/ICNN.1995.488968
- [9] Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*. 2007;28:2618-2640. DOI: 10.1002/jcc.20829
- [10] Richmond P. Template driven agent based modelling and simulation with CUDA. In: Hwu WM, editor. *GPU Computing Gems Emerald Edition*. Boston, USA: Morgan Kaufmann; 2011
- [11] Richmond P, Walker D, Coakley S, Romano D. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*. 2013;11(3):334-347. DOI: 10.1093/bib/bbp073
- [12] Vuduc R, Choi J. A brief history and introduction to GPGPU. In: Shi X, Kindratenko V, Yang C, editors. *Modern Accelerator Technologies for Geographic Information Science*. Boston, MA: Springer; 2013. DOI: 10.1007/978-1-4614-8745-6_2
- [13] Hérault A, Bilotta G, Dalrymple RA. SPH on GPU with CUDA. *Journal of Hydraulic Research*. 2010;48(Extra Issue):74-79. DOI: 10.1080/00221686.2010.9641247
- [14] Bilotta G. GPU Implementation and Validation of Fully Three-Dimensional Multi-Fluid SPH Models. *Rapporto Tecnico*. 2014:292 INGV. Available from: <http://istituto.ingv.it/images/collane-editoriali/rapporti%20tecnici/rapporti-tecnici-2014/rapporto292.pdf> [Accessed: September 05, 2018]
- [15] 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: Harris AJL, De Groeve T, Garel F, Carn SA, editors. *Detecting, Modelling and Responding to Effusive Eruptions*. Geological Society, London. 2016;426. DOI: 10.1144/SP426.24. Special Publications

[16] Zago V, Bilotta G, Hérault A, Dalrymple RA, Fortuna L, Cappello A, et al. Semi-implicit 3D SPH on GPU for lava flows. *Journal of Computational Physics*. 2018;375:854-870

[17] Zago V, Bilotta G, Cappello A, Dalrymple RA, Fortuna L, Ganci G, et al. Preliminary validation of lava benchmark tests on the GPUSPH particle engine. *Annals of Geophysics*. 2018;61. In Press

[18] GPUSPH v4.1 [Internet]. Available from: <http://www.gpusph.org/> [Accessed: September 05, 2018]

[19] Khronos OpenCL Working Group. The OpenCL™ Specification [Internet]. 2018. Available from: <https://www.khronos.org/registry/OpenCL/> [Accessed: September 05, 2018]

[20] NVIDIA. CUDA C Programming Guide [Internet]. 2018. Available from: <https://docs.nvidia.com/cuda/> [Accessed: September 05, 2018]

[21] Green S. Particle Simulation Using CUDA. NVIDIA Corporation Whitepaper. 2010. Available from: <http://developer.download.nvidia.com/assets/cuda/files/particles.pdf> [Accessed: September 05, 2018]

[22] Bédorf J, Gaburov E, Portegies Zwart P. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics*. 2012;231(7):2825-2839. DOI: 10.1016/j.jcp.2011.12.024

[23] 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*; March 2014; Paris; 2014. pp. 127-133

[24] Burtscher M, Pingali K. Chapter 6: An efficient CUDA implementation of the tree-based Barnes Hut N-body. In: *GPU Computing Gems Emerald Edition*. Boston, USA: Morgan Kaufmann; 2011. pp. 75-92. DOI: 10.1016/B978-0-12-384988-5.00006-1

[25] Morton GM. A computer oriented geodetic data base; and a new technique in file sequencing. IBM Technical Report; 1966

[26] Rustico E, Bilotta G, Hérault A, Del Negro C, Gallo G. Advances in multi-GPU smoothed particle hydrodynamics simulations. *IEEE Transactions on Parallel and Distributed Systems*. 2012; 25(1):43-52. DOI: 10.1109/TPDS.2012.340

[27] Li XS, Demmel JW, Bailey DH, Henry G, Hida Y, Iskandar J, et al. Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*. 2002;28(2):152-205. DOI: 10.1145/567806.567808

[28] Colberg PH, Höfling F. Highly accelerated simulations of glassy dynamics using GPUs: Caveats on limited floating-point precision. *Computer Physics Communications*. 2011;182:1120-1129. DOI: 10.1016/j.cpc.2011.01.009

[29] Horner WG. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*. 1819;109:308-335. JSTOR: <http://www.jstor.org/stable/107508>

- [30] Ostrowski AM. On two problems in abstract algebra connected with Horner's rule. In: von Mises R, editor. *Studies in Mathematics and Mechanics*. New York, USA: Academic Press; 2013. pp. 40-48. DOI: 10.1016/B978-1-4832-3272-0.50010-7
- [31] Pan VY. Methods of computing values of polynomials. *Russian Mathematical Surveys*. 1966;21(1): 105-136. DOI: 10.1070/RM1966v021n01ABEH004147
- [32] Kahan WM. Further remarks on reducing truncation errors. *Communications of the ACM*. 1964; 8(1):40. DOI: 10.1145/363707.363723
- [33] Neumaier A. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *Zeitschrift für Angewandte Mathematik und Mechanik*. 1974;54: 39-51. DOI: 10.1002/zamm.19740540106
- [34] Klein A. A generalized Kahan-Babuška-Summation-algorithm. *Computing*. 2006;76(3-4):279-293. DOI: 10.1007/s00607-005-0139-x
- [35] Blinn JF. Floating-point tricks. *IEEE Computer Graphics and Applications*. 1997;17(4):80-84. DOI: 10.1109/38.595279
- [36] id Software. *Quake III: Arena* [Video Game]; 1999
- [37] Domínguez JM, Crespo AJC, Barreiro A, Rogers BD, Gómez-Gesteira M. Efficient implementation of double precision in GPU computing to simulate realistic cases with high resolution. In: *Proceedings of the 9th SPHERIC Workshop*; March 2014; Paris; 2014. pp. 140-145
- [38] Hérault A, Bilotta G, Dalrymple RA. Achieving the best accuracy in an SPH implementation. In: *Proceedings of the 9th SPHERIC Workshop*; March 2014; Paris; 2014. pp. 134-139
- [39] cppreference. `std::conditional` [Internet]. 2018. Available from: <https://en.cppreference.com/w/cpp/types/conditional> [Accessed: September 05, 2018]
- [40] cppreference. `std::enable_if` [Internet]. 2018. Available from: https://en.cppreference.com/w/cpp/types/enable_if [Accessed: September 05, 2018]
- [41] Langr D, Tvrdík P, Dytrych T, Draayer JP. Fake run-time selection of template arguments in C++. In: Furia CA, Nanz S, editors. *Objects, Models, Components, Patterns. TOOLS 2012. Lecture Notes in Computer Science*. Vol. 7304. Berlin, Heidelberg: Springer; 2012. DOI: 10.1007/978-3-642-30561-0_11