Advances in Engineering Software 148 (2020) 102802

Contents lists available at ScienceDirect

Advances in Engineering Software

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

A unified object-oriented framework for CPU + GPU explicit hyperbolic solvers

Daniel A.S. Conde*, Ricardo B. Canelas, Rui M.L. Ferreira

CEris, Instituto Superior Técnico, Universidade de Lisboa, Portugal

ARTICLE INFO

Keywords:
Hyperbolic conservation laws
Parallel computing
Contextual object splitting
OMP
CUDA

ABSTRACT

A unified design solution for heterogeneous explicit hyperbolic solvers is herein introduced. The proposed design is entirely cross-compatible between CPUs and GPUs, through an intuitive object-oriented approach. The advantages of a unified CPU + GPU development approach are discussed and exemplified, and a complete description of the data and code structures are provided and benchmarked. The benefits of different object-oriented designs are quantified under static and dynamic loads in terms of parallel performance and scalability. A fair comparison with graphics processors provides a realistic measure of achievable GPU implementation benefits. Both automatically and manually tuned GPU executions are compared and shown to also have a significant impact on the obtained performance. Overall, the proposed design combines a good sequential performance with a supra-linear scalability on modern CPUs. On GPUs, execution is shown to be up to 40 times faster than its single-threaded counterpart, opening a wider range of applicable model scales and resolutions.

1. Introduction

A plethora of physical phenomena in continuum media can be mathematically represented by systems of hyperbolic conservation laws. Applications include fluid dynamics, astrophysics, electromagnetism or traffic flow. Over the last decade, several advances have been made towards more robust and well-balanced numerical methods for this class of problems, namely regarding the Euler equations [27] and the shallow-water equations [14,18].

In practice, solution correctness is not only dependent on the quality of the employed numerics, but also on the density of discretization units that compose a domain of interest. Higher densities are required in the presence of non-smooth solutions, calling for increased computational capabilities. As a result of the ever growing demands, and with processor clock speeds stalling for the past decade, the hardware industry shifted towards parallel computing, a still ongoing transition that is not without its challenges [1].

Presently, the two most used parallel devices are multi-core central processing units (CPUs) and many-core graphics processing units (GPUs). The lack of a unified programming model makes exposing parallelism remarkably conditional to the chosen architecture, often enforcing device-specific implementations. Thus, the next generation of numerical tools must consider this duality from the earliest stages of development.

The major novelty of this work is the introduction and assessment of a unified design solution for parallel and heterogeneous hyperbolic solvers. The code and data structures herein proposed are applicable to a wide range of explicit solvers, where the values of the conservative variables of a given discretization unit at some generic time step are obtained from a combination of the primitive variables from neighbouring units at the previous time step.

Over the last years, the *Open Multi-Processing* [21] and the *Compute Unified Device Architecture* (CUDA by 20) programming models were successfully employed to adapt existing hyperbolic solvers for High-Performance Computing (HPC). As an example, hyperbolic solvers for shallow-water applications were adapted to OpenMP by Zhang et al. [30] and Liu et al. [15], while in the case of CUDA the same type of solver was successfully ported to GPU devices by de la Asunción et al. [9], Lacasta et al. [12], Vacondio et al. [28] and Macías et al. [16]. By using the OP2 domain specific language, Reguly et al. [24] managed to couple both models under a CPU-GPU shallow-water code.

To avoid device-specific implementations, our suggested design is entirely cross-compatible between the OpenMP and CUDA programming models with standard C++. Additionally, and contrasting with the more common data-oriented (DO) approaches in HPC codes [26,29], we recover the intuitive object-oriented (OO) paradigm, coupled with cache-conscious decomposition, to achieve high performance levels while preserving the developer-centered features of this

* Corresponding author.
E-mail address: daniel.conde@tecnico.ulisboa.pt (D.A.S. Conde).

https://doi.org/10.1016/j.advengsoft.2020.102802
Received 18 August 2019; Received in revised form 18 March 2020; Accepted 26 March 2020
Available online 24 June 2020
0965-9978/

paradigm [11]. This is a novel approach that effectively bridges CPU and GPU development under a conceptually simple programming style, enabling heterogeneous computing capabilities and significantly reducing development overheads for additional numerical and physical features.

The present paper is organized into 6 sections. A description of the mathematical core of the targeted hyperbolic systems is presented in Section 2, followed by a technical specification of the adopted design solutions in Section 3. Section 4 analyses the impact of the suggested optimizations and benchmarks both CPU and GPU performance in problems with static and dynamic workloads. Lastly, Section 5 draws the final conclusions and prospects of future developments.

2. Non-homogeneous 2D hyperbolic systems

The present work focuses on two-dimensional finite volume solvers for hyperbolic nonlinear systems of conservation laws, in the form

$$\frac{\partial \boldsymbol{U}}{\partial t} + \vec{\nabla}\cdot\boldsymbol{E}(\boldsymbol{U}) = \boldsymbol{Q}(\boldsymbol{U}, \vec{\boldsymbol{x}}) \tag{1}$$

where t is time, $\vec{x}$ stands for the horizontal space coordinate, $\boldsymbol{U}$ is the vector of conserved (or state) variables and $\boldsymbol{E}$ is the conservative fluxes vector. The source term $\boldsymbol{Q}$ can be split as

$$\boldsymbol{Q}(\boldsymbol{U}, \vec{\boldsymbol{x}}) = \vec{\nabla}\cdot\boldsymbol{T}(\boldsymbol{U}, \vec{\boldsymbol{x}}) + \boldsymbol{S}(\boldsymbol{U})$$

with $\boldsymbol{T}$ representing source terms as non-conservative fluxes [13] and $\boldsymbol{S}$ defining only source terms that are locally determined. As a result, Eq. (1) can be rewritten as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \vec{\nabla}\cdot(\boldsymbol{E} - \boldsymbol{T}) = \boldsymbol{S} \tag{2}$$

with a finite volume formulation being obtainable by integrating (2) over a time-invariant volume $\forall$ and applying the Gauss theorem to the flux terms, resulting in

$$\frac{\partial}{\partial t}\int_{\forall} \boldsymbol{U}\, d\forall + \oint_{\partial\forall} (\boldsymbol{E} - \boldsymbol{T})\boldsymbol{n}\, dA = \int_{\forall} \boldsymbol{S}\, d\forall \tag{3}$$

where $\boldsymbol{n}$ is the outward unit vector normal to $\partial\forall$. Considering a Gudonov-type average representation of the integrated quantities, Eq. (3) can be discretized in the form

$$\frac{\boldsymbol{U}_i^{n+1} - \boldsymbol{U}_i^n}{\Delta t} A_i + \sum_{k=1}^{K_i} \left(\underset{k}{\Delta} \boldsymbol{E}_i^n - \boldsymbol{T}_{i_k}^n \right) \boldsymbol{n}_{i_k} l_{i_k} = \boldsymbol{S}_i^n A_i \tag{4}$$

where $\boldsymbol{U}_i^n$ is the state vector, averaged over volume i at time step n, A_i is the base area and Δt is the time step. Index k represents the k^{th} edge of volume i, from a total K_i, with L and $\boldsymbol{n}$ being the respective length and i-outward normal (Fig. 1a). The $\underset{k}{\Delta}$ operator denotes a quantity jump across the k^{th} edge, so that $\underset{k}{\Delta}\boldsymbol{E}_i = \boldsymbol{E}_j - \boldsymbol{E}_i$ with $j(i_k)$ being the volume adjacent to i through edge i_k. The non-conservative flux $\boldsymbol{T}_{i_k}$ is integrated across each edge i_k as defined by Murillo and García-Navarro [17], Parés and Castro [22].

The stencil in (4) is of first-order in space, suitable for discretizations in both structured and unstructured meshes (Fig. 1). The design herein described is focused on this class of stencils, although its applicability is extendable to numerical schemes with higher orders of time accuracy. Specifically, the design of the time iterator is not limited by the proposed data-structures as long as the stencil is of first order in space or reducible to first-order, e.g. by multi-step computation or recursion. The class of schemes herein explored features edge fluxes of the form $\boldsymbol{f}(\boldsymbol{U}_i^n, \boldsymbol{U}_j^n)$, where $j(i_k)$ is the k^{th} neighbor of volume i and n is the time integration step.

3. Optimization for parallel computing

In numerical modelling, the quality of the obtained solutions typically depends on the amount of discretization units that compose the computational domain. Increasing the count of these units often incurs a supra-linear penalty in the time required to compute the solution, in what is possibly the most common trade-off faced by modellers: resolution, in both time and space, *versus* available computation time.

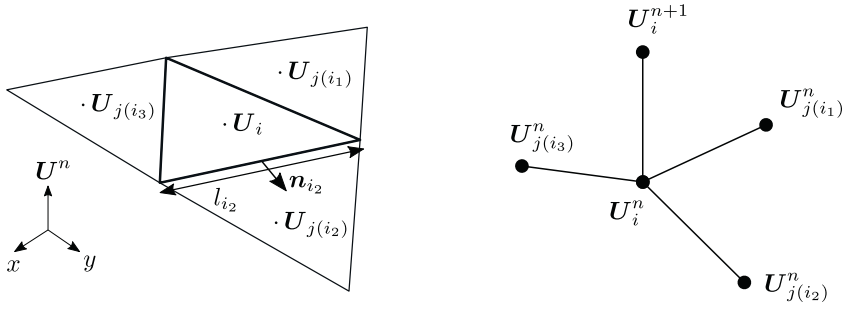
The constant demand for computational power is, of course, one of the driving forces of the hardware industry. Over the last decade, as power consumption stalled the otherwise escalating clock frequencies, hardware design has shifted towards more (rather than faster) *arithmetic units* on both CPUs and GPUs.

The main architectural differences between these two devices are shown in Fig. 2: modern CPUs have larger cache memories and sophisticated units for control (CU) and arithmetics (ALU), whereas GPUs favour simpler and more numerous ALUs. Consequently, CPUs excel at multiple data processing with multiple instructions (MIMD), while GPUs are optimized for batch processing multiple data with single instructions (SIMD). Different instruction sets are thus used on CPUs and GPUs, respectively the `x86` and `PTX` sets, requiring the usage of different *Application Programming Interfaces* (APIs) to exploit parallelism with these architectures.

Since no truly unified API exists for joint CPU-GPU programming, we herein propose a new code design for parallel hyperbolic solvers, based on `C/C++` coupled with two APIs – *Open Multi-Processing* [21], for parallel CPU execution, and *Compute Unified Device Architecture* (CUDA by 20) for GPUs. The particular topics discussed are generic to most hyperbolic solvers, being herein applied and tested on hydrodynamics applications.

3.1. Unified source code design

Developers, particularly those working with intensive applications, are faced with many challenges when moving towards a parallel computing paradigm: from data consistency to race conditions, mutual exclusions, synchronization and load balance, among others. Consequently, parallel programs become much more complex than their sequential counterparts.



(a) Volume connectivity (b) Computational stencil

Fig. 1. Required variables for flux computation and time integration of volume states.

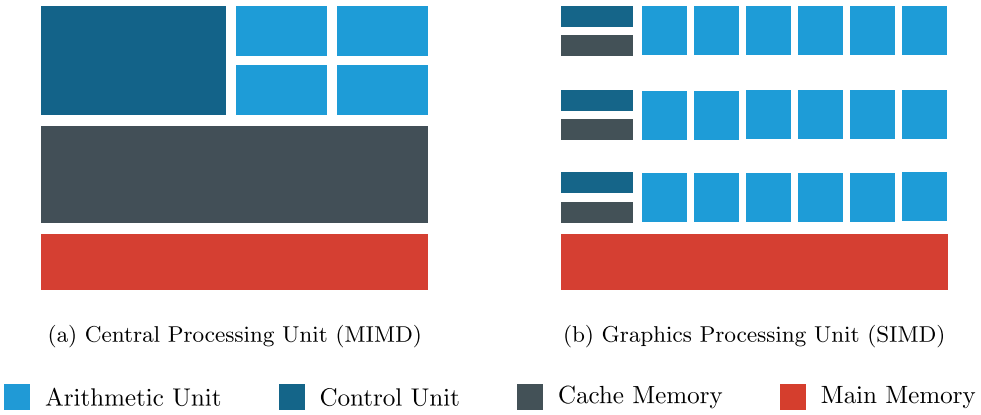


Fig. 2. Architectural differences between CPUs and GPUs.

Avoiding such cluttering is a specific goal in the proposed code design: an attempt to balance parallel performance, code productivity and CPU + GPU compatibility. Our `C/C++` source code is fully modularized and divided into the following main groups:

- **Global** and control variables,
- **Mesh** entities and geometry,
- **Boundary** and initial conditions,
- **Hyperbolic** solver(s),
- **Local** source terms,
- **External** forcing,
- **Storage** input/output,
- **GPU** input/output,
- **GPU** kernels.

The code design is said to be *unified* in the sense that the compilation units used for both CPU and GPU compilation are largely the same. The unavoidable exceptions are the two GPU-specific source files stated above, which constitute less than 10% of the total code. Namely, the *GPU input/output* unit concentrates CUDA's memory management calls, required to allocate, deallocate and copy simulation data to and from the GPU. The second GPU-specific file declares a set of specific CUDA functions called *kernels*, which are the work units of GPU parallelism and are executed simultaneously by hundreds of threads.

3.1.1. CPU implementation

In the targeted explicit solvers, each stage requires invoking some procedure repeatedly over a large set of units. On the CPU this is performed via typical `for` cycles, where each thread executes a subset of these units. OpenMP provides `#pragma` clauses to signal code regions that are to be compiled with multi-threaded support, admitting additional attributes to specify data sharing, thread synchronization and scheduling.

The data-structure, described in the following sub-section, was designed for full iteration independence and thus doesn't require sharing directives. Parallelization under this API then becomes a minimal effort: parallel cycles differ from their sequential counterparts by a simple one-lined `#pragma` directive, specifying the work-sharing construct type and configuration (Listing 1 and CPU code in Fig. 3).

In detail, a `team` of OpenMP threads (Fig. 3) is created (forked), by invoking a `omp parallel` region, which completely encompasses the `while` time cycle, as shown in Listing 1. The work-sharing constructs `omp for` are then used inside this `parallel` region to distribute each `for` cycle's iterations by this `team`, according to a specified `schedule` option. A common (and cleaner) alternative would be to suppress the initial `omp parallel` invocation and locally specify a combined `omp parallel for` construct on each cycle. However, this would create and destroy thread `team`s on a per-cycle basis, introducing unnecessary thread management and synchronization overheads [21].

After flux computation (see Listing 1), the global time step (`cpuNumerics.dt`) is obtained by reducing the thread-specific time steps (`threadMinDt`) taken as the allowable minimum according to all fluxes computed by each thread. This is forcefully a `critical` code segment, in which all threads enter and leave sequentially. Contrarily to the `omp for` constructs, no implicit synchronization occurs after a `critical` section [21], meaning that all threads must be explicitly synchronized by means of a `barrier`. Again, a cleaner alternative would be to add a reduction-to-minimum clause to the fluxes `omp for` clause, such as `reduce(min:dt)`, although this reduction feature is not equivalently supported by CUDA kernel launches and was therefore not considered.

The following stages shown in Listing 1, namely state update, sources and forcing computations, programatically differ from the fluxes stage only in the sense that synchronization is not required: threads always operate on the same volumes and these three stages are locally determined. The `nowait` attribute is thus specified on their `omp for` clauses removing all implicit thread synchronizations after these cycles.

The division of cycle iterations among OpenMP threads, which is specified by the `schedule()` attribute (see Listing 1), has a direct impact on parallel scalability. In solvers with sparse workloads, where not all volumes have permanent non-zero states, some threads will take longer to compute than others. A `static` scheduling, defined at compile-time, binds threads to invariant and potentially contiguous memory ranges, with common options including

1. `schedule(static)`: divides volumes among threads in contiguous, equally-sized blocks,
2. `schedule(static,1)`: completely interleaves volumes among threads
3. `schedule(static,b)`: interleaves contiguous, `b`-sized volume blocks among threads.

The first option results in load unbalance for scenarios with a large ratio of inactive to active cells, since some threads will be assigned significantly more wet volumes and will take longer than the rest. This is solved by the nearly isotropic distribution of threads across the domain in option 2, but with a strided access pattern that degrades memory performance. Option 3 provides the best configuration if the `b` parameter is carefully weighted: large enough to avoid a permanently strided access and small enough to isotropically distribute threads across the domain.

This provides a better solution for dealing with workload sparsity, relatively to the alternative `dynamic` scheduling which adjusts the workload of each thread during run-time at the cost of a thread monitoring overhead. A visualization of the `b` parameter and an analysis of

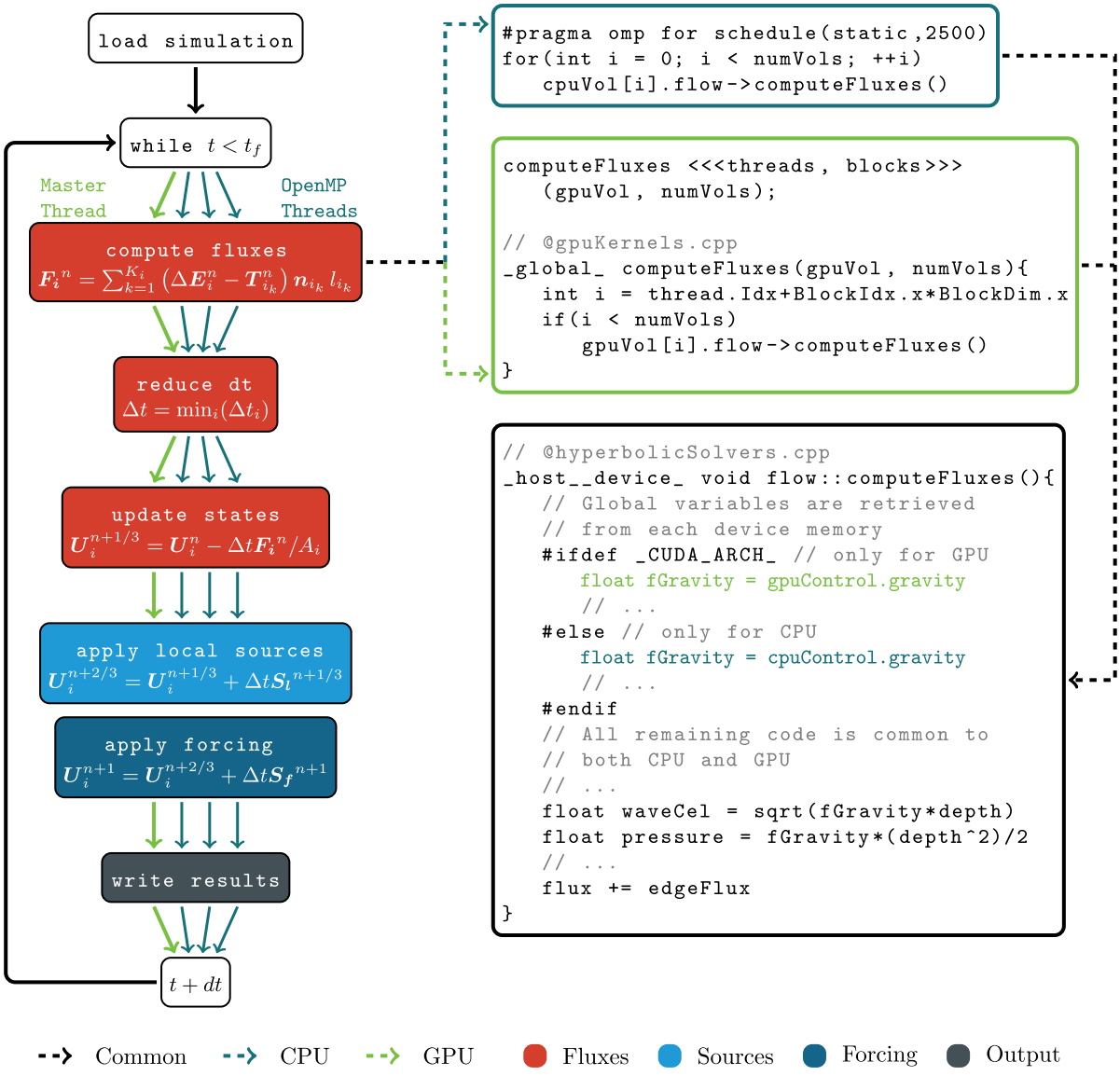


Fig. 1. OpenMP parallel regions in the main CPU cycle.

its influence on scalability is provided in Section 4.3.

3.1.2. GPU implementation

In CUDA code, the `omp for` constructs and cycles in Listing 1 are simply replaced by GPU kernel calls of the form `kernel<<< · , · >>> ()`, as shown in Listing 2, complemented by short kernel declarations in a separate `gpuKernels.cpp` compilation unit (see Fig. 3).

In order to maximize speedup, the data transfers between CPU and GPU must be minimized. After the mesh is initially copied to the GPU, all simulation stages are entirely performed on this device (Listing 2). Hence, because the CPU still controls kernel launches and time progression, only a single systematic `cudaMemcpyToSymbol` transfer of the time step (8 bytes) is required to synchronize the two devices.

Examining the flux computation code samples in Fig. 3, one can verify that GPU and CPU codes only differ by a few lines of code, since the core of the computation resides in the hybrid `flow::computeFluxes()` method. Most of the code is thus shared between both devices, only differing in the syntax for exposing parallelism, which is deeply tied to each API's specification. For the shown example, 97.7% of the executed code is shared between CPU and GPU (220 lines out of 224), and all remaining stages of the computation follow exactly the same line of reasoning.

This is of paramount importance for the maintainability and future augmentation of heterogeneous codes: corrections and new features can be conventionally programmed and debugged in CPU development environments, while the code remains readily compilable for GPU execution. The key of this cross-compatible design are the hybrid CPU + GPU object methods, which must conform to five rules, some of them deriving directly from the CUDA specification [20]:

1. **Object members** must contain only plain data types,
2. **Object polymorphism** is, generally, not allowed,
3. **Object methods** must be declared with both `_host_` and `_device_` compiler flags,
4. **Object methods** are self-contained: they only modify their own members,
5. **Object methods** only access global variables in the device they are allocated.

First and second rules are direct consequences of the programming language in CUDA: an extension of `C`. Hence, the more sophisticated types provided by `C++`, like virtual classes and functions, are not efficiently portable to GPU code. Conveniently, and as long as the involved members are `C`-compatible types, classes are still allowed to contain,

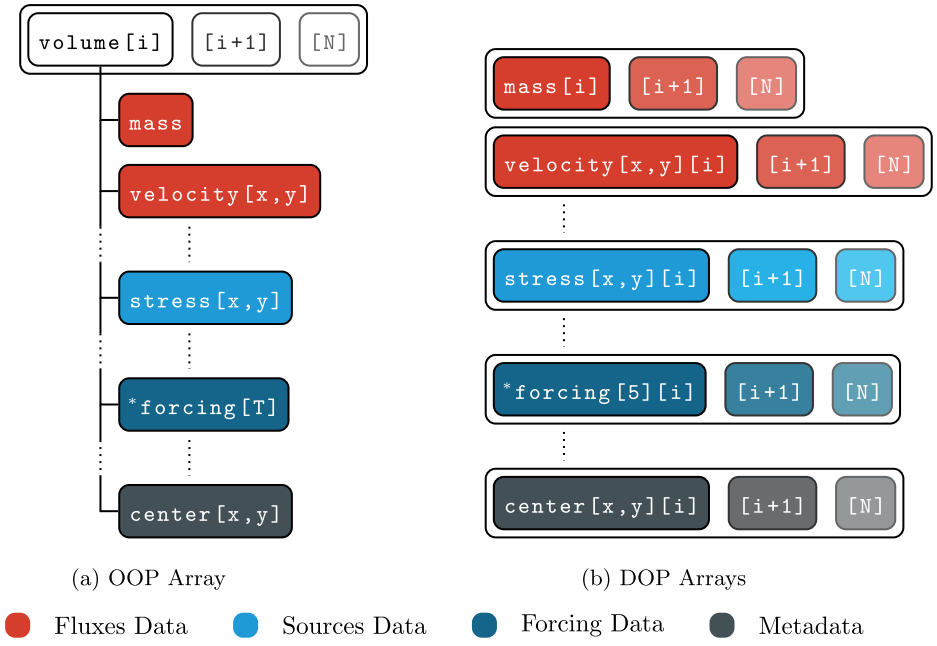


Fig. 2. CUDA kernel calls in the main GPU cycle.

friend and inherit other classes.

The third rule signals the double compilation of the method: one for `x86_64` instructions (CPU) and another one for the `PTX` instruction set (GPU). Both versions of the method are then linked to the main executable and may be used concurrently.

The fourth rule is common to both CPU and GPU parallelization: if all computations can be reduced to methods that exclusively modify their object's members, then the problem becomes *embarrassingly parallel* and one must only divide object arrays across available OpenMP/CUDA threads.

Global variables are addressed in the fifth rule. Because the same code is compiled for two devices with disjoint global memories, a fork must be introduced to direct the compiler to reachable variable references. In nVidia's CUDA compiler the specific macro `_CUDA_ARCH_` signals `PTX` compilation. A `#ifdef` pre-processor directive uses this macro to use the global variables allocated on the targeted device's memory (either cpuControl or gpuControl as in the example displayed in Fig. 3).

The kernel calls use two parameters to specify the launch configuration on the GPU, e.g. `computeFluxes <<<threads, blocks>>> (volume, numVol)` in Listing 2 and Fig. 3. The chosen number of threads and blocks affects raw GPU performance in a highly non-linear way, while at the same time playing a role comparable to the `b` parameter in OpenMP. This is analyzed and discussed for each stage's kernel in Section 4, on par with the OpenMP load balancing.

3.2. Data structure design

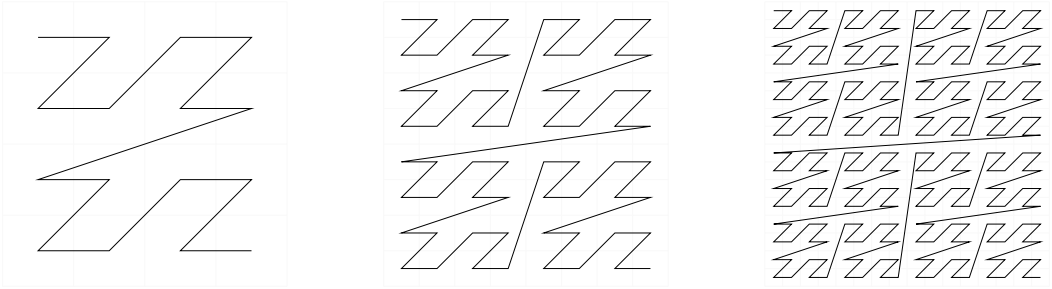
3.2.1. Object-oriented programming

The proposed code and data-structure design is largely supported by the paradigm of Object-Oriented Programming (OOP). In this design, an *object* is an individual volume and all computations are implemented as *methods* operating on *only that* volume. This is an extremely important concept in OOP called *encapsulation* which renders the code trivially parallelizable on both CPUs and GPUs. Several characteristics of OOP make it the dominant programming methodology in Computer Science, where the most notable ones are code reusability, maintainability, simplicity and, consequently, better design options for larger programs. However, and despite OOP's popularity, it is still seldom used in HPC.

3.2.2. Memory hierarchy and data locality

Computer memories are highly hierarchic: the data channel feeding the processor has multiple levels of decreasing access times but also decreasing size, where the *cache* memory, built within the processor itself, is the fastest in the hierarchy. It provides a narrow, high-speed buffer of the *main memory*, which is typically two orders of magnitude slower to access [2].

In HPC applications, developers have to exploit this hierarchy by designing programs that favour *locality of reference*: memory locations accessed close in time are likely to be close in memory. Accesses to the main memory are thus kept to a minimum, since most of the data



(a) $S = [0,3] \times [0,3]$ (b) $S = [0,7] \times [0,7]$ (c) $S = [0,15] \times [0,15]$

Fig. 3. Main cycle and code samples for a hydrodynamics solver. The sources S are split into S_l and S_f, respectively standing for locally determined or externally forced terms.

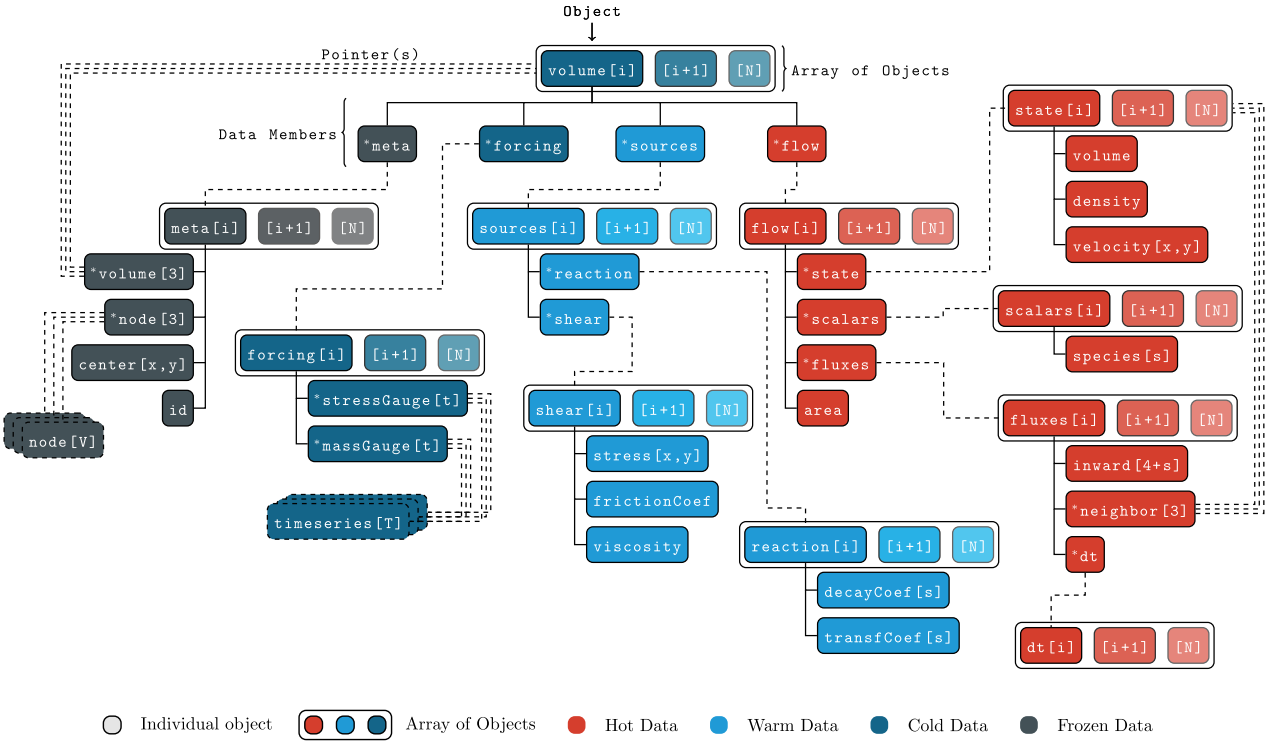


Fig. 4. Example of two memory-layout options for a fluid dynamics application.

required by the processor at a given time will likely be *cached*. The main argument supporting the avoidance of OOP in HPC is that it typically produces programs with very poor data locality.

A naive memory arrangement of an Array of Objects (AoO) for a fluid dynamics example is shown in Fig. 4a, where each object (volume) contains a set of fields and is allocated sequentially. The advantages of this design are an intuitive grouping of the information and a much simplified memory management, as is typical of OOP. The downside is that the spacing of the same field across different objects is large and, when a method accesses only a few fields, the cache will be populated with unworkable data. This is known as *cache pollution* and severely compromises its purpose.

As a result, most compute-intensive applications in modeling, machine learning and gaming usually resort to Data-Oriented Programming (DOP). In this approach, each field exists in a Discrete Array (DA, Fig. 4b), which comprises a linear and homogeneous data-structures that is inherently cache-friendly.

However, the ability to encapsulate data and procedures is lost, and the complexity of memory management is increased by a factor equal to the total number of fields. This would mean that data allocation, deallocation and CPU-GPU memory management would require significantly more coding with DA/DOP, when compared to OOP.

3.2.3. Contextual object splitting (CoS)

Performance and code simplicity are of utmost importance for code longevity. In this sense, we herein propose a design that preserves the coding and maintenance conveniences provided by object-oriented designs while still achieving a comparable performance to data-orientation.

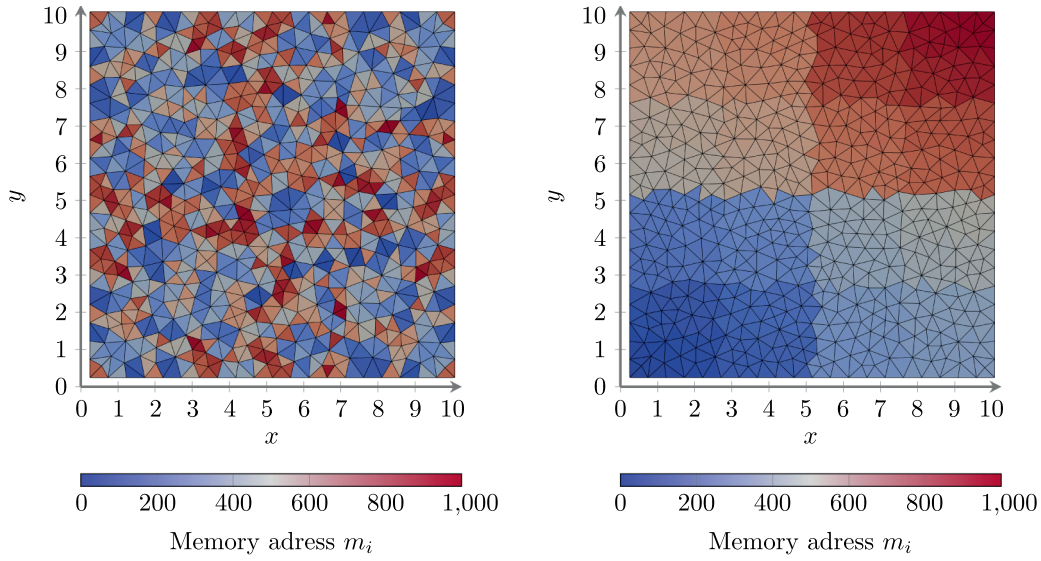
Instead of fully integrating volume-wise quantities into a single array of objects (AoO) or, oppositely, fragmenting all this data as discrete arrays (DA), our proposal is that quantities accessed close in time should form a sub-object and become encapsulated into what we called *contexts*. Essentially, the mono-block design shown in Fig. 4a is split into multi-blocks, whose fields are always accessed together by context based methods. These *contexts* are then linked by means of *pointers*, which, from the developer's standpoint, emulate the centralized design of object-orientation. This is essential to preserve data-method encapsulation, while still avoiding the performance issues related to cache pollution: contexts are only brought to cached when their data is required.

Following this method, Fig. 5 displays an example of a data-structure for a fluid dynamics example. Four main contexts were used to split volume-wise data throughout the code: the `flow` context, containing fluid states, scalars and fluxes, the `sources` context, with all the quantities required for locally determined sources or sinks, the `forcing` context, containing information regarding uncoupled source terms, and lastly the `meta` context, comprising meta-data mostly used during pre/post-processing and file output.

The color-coding in Fig. 5 introduces data usage frequency as a relevant aspect for cache-conscious object splitting [5]. In common terms, the usage frequency can be figuratively defined as data *temperature: hot*, for heavily accessed data during most of the execution; *warm*, for data accessed in medium and frequent code regions, *cold* for data accessed frequently in small code sections and *frozen* for seldom accessed data in infrequent code regions. As contexts become *hotter*, they are fragmented into smaller sub-objects (Fig. 5), thus approaching the data-oriented design for intensive stages, whereas *warm* and *cold* contexts are used in decreasingly intensive stages that require less data modularity.

The `flow` context (Fig. 5) has 4 sub-contexts: `state` and `scalars`, containing the conserved variables (and passive scalars) $\boldsymbol{U}$ in (4); `fluxes`, including the $j(i_k)$ connectivity and incoming fluxes $\underset{k}{\Delta}\boldsymbol{E} - \boldsymbol{T}$ in (4); and `dt` representing Δt in (4). The locally-determined `sources` context contains the $\boldsymbol{S}$ terms in (4) and is primarily split into two sub-contexts, namely `shear` stress terms and scalar `reaction` coefficients. Sparser `timeseries` are used to implement external mass and momentum `forcing` terms, a much less intensive context that requires no further splitting. Finally, the larger `meta` context is only used when reading or writing to storage which, in applications, happens at a much lower rate compared to solver frequency.

The purpose of this effort is to avoid cache pollution by enhancing



(a) Unordered mesh (b) Z-ordered mesh

Fig. 5. Contextual object splitting (CoS) method for a fluid dynamics application.

the modularity of the data-structure, bringing it closer to the *locality of reference* principle. Splitting larger objects by usage context and frequency results in cache-friendly access patterns, that are in line with the memory hierarchy and are essential to both sequential and parallel performance, as discussed in Section 4.

3.2.4. Memory traversal

The data-structure design in Fig. 5 takes advantage of encapsulation to ensure that each `volume` only *manipulates* its own contexts. However, this does not mean that contexts of neighboring volumes are not *accessed*. Specifically, the `fluxes` context of an element i requires the `state` context of its `neighbor`ing volumes $\boldsymbol{U}_{j(i_k)}$ (Fig. 1b) to compute the `inward` fluxes associated to $\underset{k}{\Delta}\boldsymbol{E} - \boldsymbol{T}$ in (4).

When working with unstructured meshes, the inherent randomness in mesh generation will likely place neighbouring volumes in 2D space far apart in the 1D memory space. This causes a different form of cache pollution, where unworkable data is brought up to the cache due to a spatially incoherent object *ordering*, instead of a cache-oblivious object *layout* as is the case of an AoO.

Space-filling curves prescribe coherent transformations between spaces of different dimensionality [4]. For the problem at hand, a space filling curve can be defined as a $\mathbb{N}^2 \rightarrow \mathbb{N}$ mapping from the Euclidean 2D-plane (x, y) to the 1D discrete memory space M. The particular curve employed was a Z-Order curve [8], named after the trajectory it follows when traversing a domain $S \in \mathbb{N}^2$, as shown in Fig. 6.

The Z-coordinate of a volume is obtained by interleaving the bit representations of the barycenter,

$$(Z_2)_b := \begin{cases} (Y_2)_{b/2} & \text{if } b \text{ is even,} \\ (X_2)_{(b+1)/2} & \text{if } b \text{ is odd,} \end{cases} \tag{5}$$

where $(\cdot\,)_2$ denotes bit representation, b is the bit index and (X, Y) are positive and rounded coordinates equivalent to (x, y). The i index of a given volume, and consequently its memory address m_i, is retrieved from the ascendant sorting of all Z-coordinates. The result is a spatially coherent memory access pattern that exhibits high data-locality: when processing some volume i, it's neighboring volumes $j(i_k)$ are in the vicinity of i itself. This is highlighted in Fig. 7 where, with the Z-ordering, one can verify that neighboring volumes become adjacently allocated in memory.

For triangular meshes, a qualitative measure of cache performance can be given by the average distance in memory, $\bar{\xi}$, between each volume and its 3 neighbors,

$$\bar{\xi} = \frac{1}{3N} \sum_{i=1}^{N} \sum_{k=1}^{3} |m_i - m_k| \tag{6}$$

where lower values mean better locality. This is shown in Fig. 8, where each volume i is plotted with the neighboring volume indexes $j(i_k)$. The line represents the ideal case where $j(i_k) = i \pm \epsilon$, with $\epsilon \in \{1, 2, 3\}$.

A randomly ordered mesh like the one in Fig. 8a implies *cache-misses* in almost every computation. The large distance in memory between volumes and neighbors, $\bar{\xi} = 420$, prevents this ensemble from ever being *cached* simultaneously. In contrast, in the Z-ordered mesh all volumes and neighbors are close in memory, $\bar{\xi} = 9$, and the likelihood of consecutive *cache-hits* is very high. This is visible in Fig. 8b, where the distance between the dots and the ideal line is minimal and the addresses of the dots is continuous.

4. Performance analysis

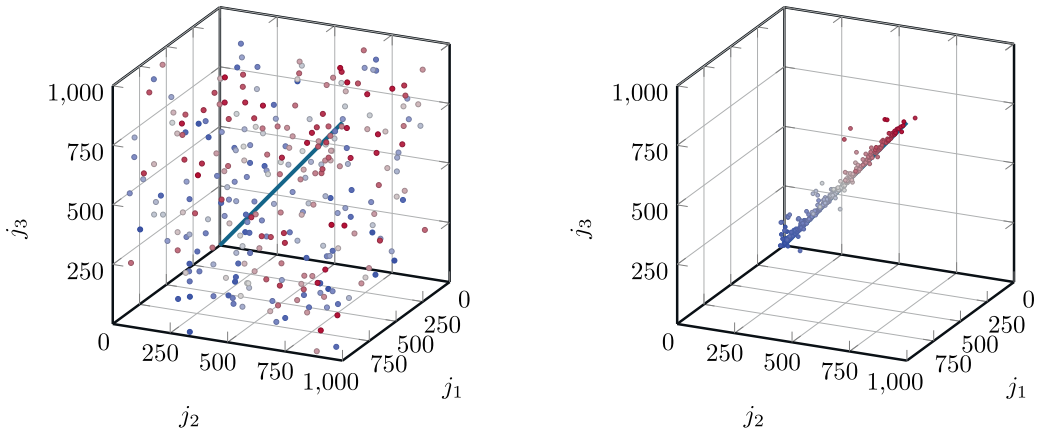
4.1. Application to a shallow-water model

This design is composed of multiple steps, deriving from the code and data-structure features described in 3.2, which could be implemented individually on existing finite-volume codes. The performance benefits of each incremental step are herein quantified with synthetic tests under static and dynamic loads.

As a working example, the former code *Strong Transients in Alluvial Valleys* (STAV, by Ferreira et al. [10], Canelas et al. [3] and Conde et al. [6]), a Roe-Riemann solver for water-related decision support systems, was fully reworked for the HPC age using the proposed design (Hi-STAV).

The specific hyperbolic conservation laws upon which Hi-STAV operates are the shallow-water equations. In this framework, the Navier-Stokes system with scalar transport is vertically integrated between the bed and the free surface, leading to the following depth-averaged form of the conservation laws for total mass, horizontal momentum and scalar transport of S species

$$\partial_t h + \partial_x(hu) + \partial_y(hv) = \dot{h}_p - \partial_t z_b \tag{7}$$



(a) Unordered connectivity (b) Z-ordered connectivity

Fig. 6. Z-curves for a different number of uniformly spaced nodes.

$$\partial_t(hu_\alpha) + \partial_{x_\alpha}(hu_\alpha^2) + \partial_{x_\beta}(hu_\alpha u_\beta) = -\partial_{x_\alpha}(gh^2/2) - gh\partial_{x_\alpha} z_b + (\tau_w - \tau_b)_{x_\alpha}/\rho \tag{8}$$

$$\partial_t(\zeta_s h) + \partial_x(\zeta_s hu) + \partial_y(\zeta_s hv) = \phi_s \tag{9}$$

where h is the flow depth, z_b is the bed elevation, $u_\alpha\,(\alpha = 1, 2)$ is the depth-averaged velocity in the x_α direction, g is the gravitational acceleration and ζ_s is the depth-averaged concentration of a passive scalar s. In Eq. (8), τ_{b,x_α} and τ_{w,x_α} are the wind and bed shear stresses, respectively, and $gh\partial_{x_\alpha} z_b$ represents the bottom slope contribution. The local source terms for total mass are the bed evolution $-\partial_t z_b$ and precipitation intensity $\dot{h}_p$, while for the scalar transport laws are specific ϕ_s terms that depend greatly on their physical nature (p. ex. sediments, dissolved pollutants, temperature, etc.).

This system of $3 + S$ equations, namely (7) through (9), can be compactly rewritten in the forms of (1) or (4) by using

$$\boldsymbol{U} = \begin{pmatrix} h \\ hu \\ hv \\ h\zeta_1 \\ \vdots \\ h\zeta_S \end{pmatrix} \quad \boldsymbol{E} = \begin{pmatrix} hu & hv \\ hu^2 + gh^2/2 & hvu \\ huv & hv^2 + gh^2/2 \\ hu\zeta_1 & hv\zeta_1 \\ \vdots & \vdots \\ hu\zeta_S & hv\zeta_S \end{pmatrix} \quad \boldsymbol{T} = -gh \begin{pmatrix} 0 & 0 \\ \Delta z_b & 0 \\ 0 & \Delta z_b \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{pmatrix}$$

$$\boldsymbol{S} = \begin{pmatrix} \dot{h}_p - \partial_t z_b \\ (\tau_w - \tau_b)_x/\rho \\ (\tau_w - \tau_b)_y/\rho \\ \phi_1 \\ \vdots \\ \phi_s \end{pmatrix}$$

where $\boldsymbol{U}$ is the conserved variables vector, $\boldsymbol{E}$ are the hydrodynamic fluxes, $\boldsymbol{T}$ accounts for bed slope as a non-conservative flux, Δz_b is the bed step, and $\boldsymbol{S}$ is the sources vector. Further details on the

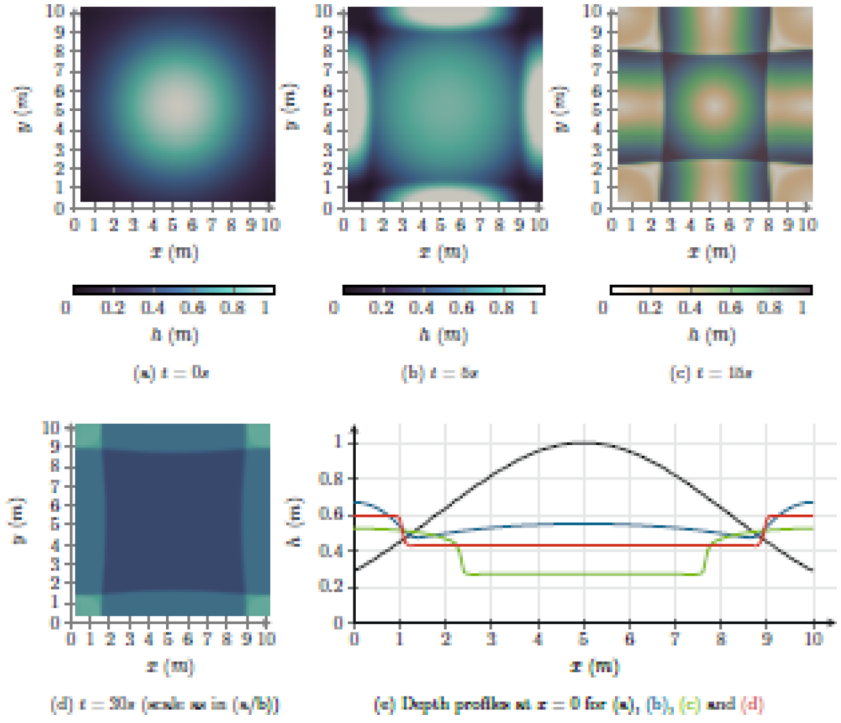


Fig. 7. Mesh coherence between physical and memory spaces.

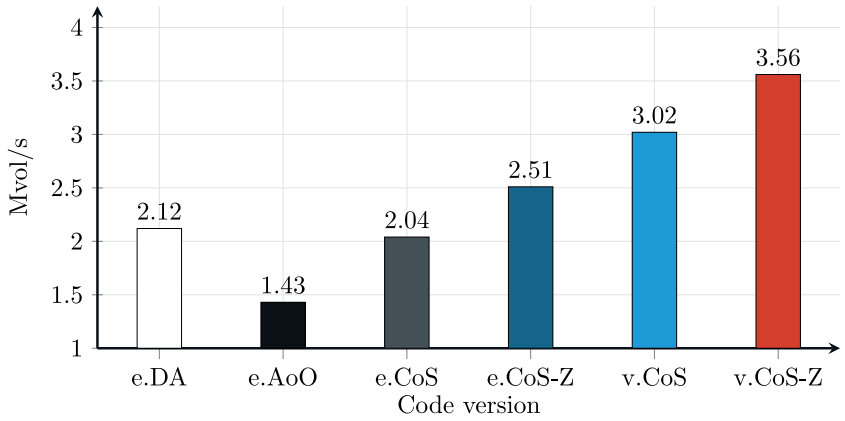


Fig. 8. Effect of Z-Ordering on mesh connectivity (memory address follows the same scale of Fig. 7).

discretization of this system are provided in the Appendix.

The time step is restricted to a conventional Courant-Friedrichs-Lewy (CFL) condition on each edge i_k, asserting that no interaction between fluxes propagating from adjacent volumes ever occurs,

$$\Delta t_{i_k} \leq \frac{A_i}{\max_m(|\bar{\lambda_m}|)_{i_k} l_{i_k}} \frac{2}{3} C_{\max}, \quad \Delta t = \min_{i_k}(\Delta t_{i_k}) \tag{10}$$

where $C_{\max}$ is the maximum Courant number, λ_m is the mth characteristic of M conserved variables and (10) is valid for triangular meshes with moderate aspect ratios and skewness. Correction mechanisms regarding entropy, mass positivity and wetting-drying fronts complete the set of stability and feasibility conditions [17] when re-averaging the solution over each volume i.

It should be noted that these synthetic tests do not configure a numerical validation attempt, but rather a performance characterization in terms of computation times. Regarding model validation, Hi-STAV has been validated with analytical, laboratory and historical data in previous efforts by Canelas et al. [3], Conde et al. [6], 7]. The re-designed code adheres strictly to the numerical scheme presented and validated in the aforementioned publications. Also, the performance metrics featured in this section were obtained from fully compiler-optimized executables (`gcc/nvcc -O3`) of Hi-STAV.

4.2. Static load test

The static test case is an all-wet simulation, preventing load-imbalances from biasing the performance analysis of the data-structure. The setup (Fig. 9a) is an initial-value problem on a $[0, 10] \times [0, 10]$ m^2 domain, configured from a Gaussian 2D free-surface on a flat bed and initially null velocity

$$h_i = \exp\left(-\frac{(x_i - 5)^2 + (y_i - 5)^2}{20}\right), \quad z_{b_i} = |\boldsymbol{u}_i| = 0$$

which results in a simulation with constant workload, since bed source terms are always null and entropy corrections are employed evenly across the domain due to multiple sharp shocks, as shown in Fig. 9. The data-structure options discussed in Section 3.2 are summarized in Table 1.

The baseline version, *e.bDA*, is a data-oriented edge-based solver [3], where all volume-wise data is organized as in Fig. 4b. The edge-based OO version, *e.AoO*, is a simple re-design of *e.bDA* according to Fig. 4a. Version *e.CoS* marks the adoption of the Contextual Object Splitting method (Fig. 5), with a volume-based variant also being produced in version *v.CoS*. Z-Ordering is considered as a separate pre-processing optimization and not a structural code design feature. Fig. 10 shows the sequential performance of each incremental code optimization, in terms of millions of volumes per second (MVol/s), on a `i7 7700k`.

As expected, the slowest version was the one implemented in a (naive) object-oriented design (*e.AoO*), showing a significant 33% performance penalty relative to the baseline and backing up the argument that OOP is inefficient for intensive applications. Switching to contextual objects with (*e.CoS*) brought the performance back to the levels of a data-oriented design. A better cache performance was obtained with Z-Ordering, *e.CoS-Z*, providing an additional 23% performance gain.

However, the strongest performance improvement, around 45%, came from adapting the solver to volume-wise operations, *v.CoS*, instead of the most common edge-wise approach in unstructured meshes [12]. The increased efficiency in memory access from using a single data-structure positively outweighs the slightly larger number of computations required by element-based solvers. Similarly to the edge-based solver, a Z-Ordering of the mesh increased performance by 18% (*v.CoS-Z*).

Fig. 11 highlights the importance of a cache-conscious design for light stencil computations, such as those in explicit hyperbolic solvers. The different memories along the hierarchy provide widely non-uniform access times and even incremental improvements to cache efficiency can provide large benefits. The average number of cycles required per memory access, C_m, is given by

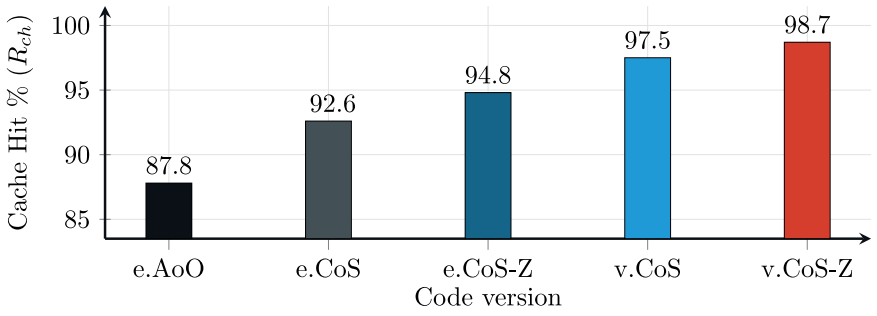
$$C_m = R_{ch}\, C_{ch} + (1 - R_{ch})\, C_{RAM}$$

where R_{ch} is the cache hit rate, which is the ratio of cached to total memory accesses, C_{ch} is the number of cycles required to retrieve data from the cache and C_{RAM} is the number of cycles required to retrieve data from the main memory, which for the tested `i7` unit are respectively 4 and 250 cycles. We herein refer to first-level caches (L1), the fastest and therefore most relevant level in terms of memory access times.

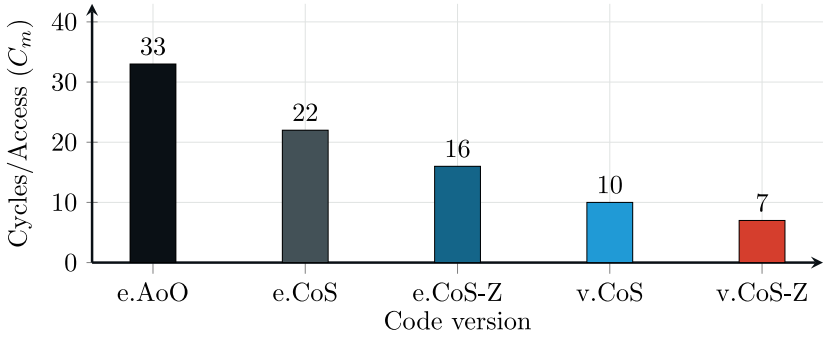
Consequently, the *e.AoO* version takes 470% longer to access memory and 248% more time in total, when compared to the fully optimized *v.CoS-Z*, even though the latter has a cache hit-rate that is better by only 11%. Comparing both volume-wise versions shows that a 28% decrease in access time and 18% in total time is obtainable from a mere 1.2% increase in the cache-hit rate.

The better efficiency of some versions is also noticeable in terms of scalability, as a good use of the cache translates to a lower occupation of the limited main memory bus. Executing several threads requires more data accesses, which may inhibit scalability due to a memory-bounded thread performance due to bus saturation. Fig. 12 shows the obtained speedup, s, relatively to the number of OpenMP threads, T_{omp}, for each version. A design with larger objects, as *e.AoO*, will saturate the memory bus with very few threads, leading to poor scalability, while a cache-conscious design, like version *v.CoS-Z*, allows a nearly ideal scalability on the tested quad-core `i7 7700k` unit.

An interesting case of 'supra-linear' speedup is noticeable in Fig. 12



(a) Cache hits as a percentage of total cache accesses



(b) Average number of cycles per each memory access, C_m

Fig. 9. Solution of the static-load test case.

Table 1
Code versions with respective incremental optimizations.

Code version	Data-structure type	Solver type
e.bDA	Discrete Arrays (baseline, bDA)	Edges (e)
e.AoO	Object-Oriented (AoO)	Edges (e)
e.CoS	Contextual Object Splitting (CoS)	Edges (e)
e.CoS-Z	Contextual Object Splitting & Z-Ordering (CoS-Z)	Edges (e)
v.CoS	Contextual Object Splitting (CoS)	Volumes (v)
v.CoS-Z	Contextual Object Splitting & Z-Ordering (CoS-Z)	Volumes (v)

for the *CoS* designs. The `i7 7700k` has 4 physical cores and is equipped with *Hyper-Threading* (HT), allowing two streams of threads to be assigned to each core: when one stream is waiting for data the other stream is executed. This technology is able to mask the cache-hit rate to some degree, since a thread that incurs in a cache-miss can be put on hold while some other thread that is ready can be executed. Even with moderate loads, our data-structure takes advantage of this technology to provide extra computational throughout, ranging from 15% on the fully optimized *v.CoS-Z* version to 4% on version *e.AoO*.

Performance on GPUs was tested with a `GeForce GTX 1080` graphics card. Comparing GPU and CPU performance is not straightforward, since a wide range of both devices is currently available, offering largely different processing capabilities. The speedup of massively-parallel CUDA implementations is normally quantified in terms of sequential performance on CPU. Hence, a *realistic* assessment of GPU adaptation benefits can only be performed by using equivalently ranked devices in these two categories, although this is not often found in the literature. In this case, the chosen pair of devices are equivalent: the `i7 7700k` is among the fastest sequential CPUs on the market (4th), while

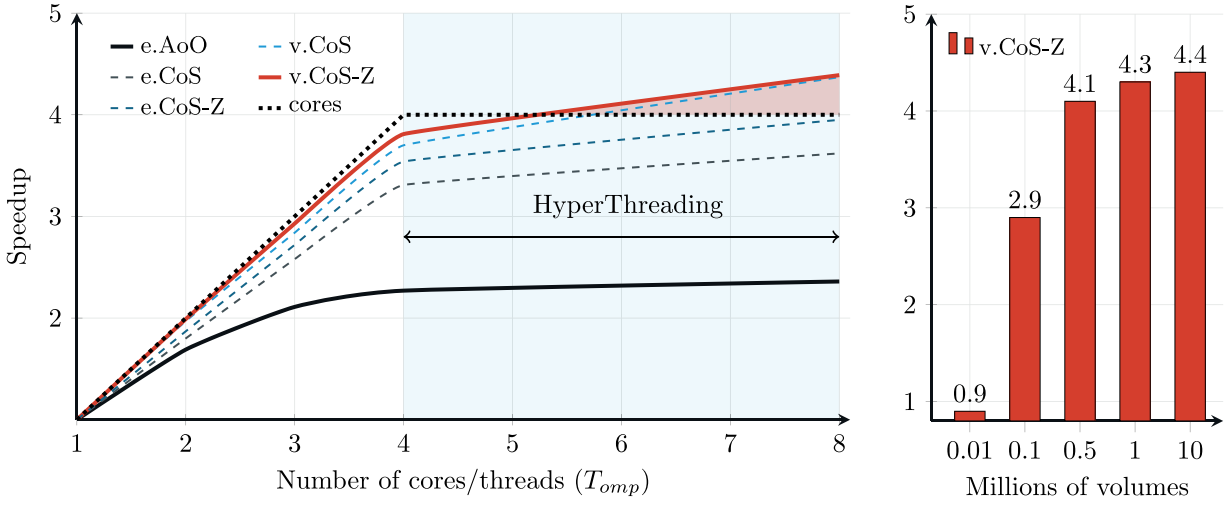


Fig. 10. Sequential performance of the different code designs.

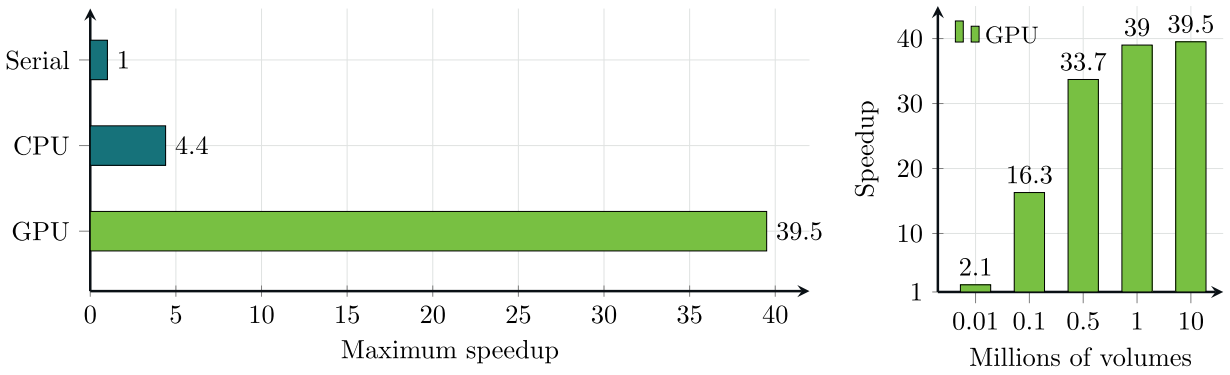


Fig. 11. Cache performance of the different code designs.

the same is true for the `GeForce GTX 1080` in terms of parallel GPU performance (5th, as tested by PassMark [23]).

Fig. 13 shows that, as expected, execution on GPU was much faster than on CPU, even for small meshes. Relatively to the sequential execution on the `i7`, CUDA execution on the `GTX` achieved a maximum speedup of 39.6. When compared to the parallel OpenMP execution with 4 cores with HT, a still considerable speedup of 9.0 was obtained.

4.3. Dynamic load test

As stated in Section 3, load balancing is also a relevant aspect of parallel efficiency. This issue was addressed by interleaving contiguous volume blocks of size b, providing an homogeneous distribution of all threads throughout the domain, with a still contiguous memory-access pattern.

This test is very similar to the static load test, but with a taller and origin-centered Gaussian surface

$$h_i = 16\exp\left(-\frac{x_i^2 + y_i^2}{5}\right), \quad z_{b_i} = |\boldsymbol{u}_i| = 0$$

having the same total water mass while also exposing dry bed regions, as shown in Fig. 14a. Two thirds of the domain are initially dry, with 0.8s being required for the flow to completely cover this area (Fig. 14b), after which it converges to an almost static load.

The effect of the `b` parameter on thread distribution throughout the domain is shown in Fig. 15. In the case of a large value for this parameter, for instance N/T_{omp} where N is the number of volumes, the domain will be split in few equally sized and contiguous blocks. The majority of the wet volumes (Fig. 15a) may thus be contained in only a few of these blocks (Fig. 15b), meaning that the computational workload will be unevenly distributed, with some threads being assigned more wet volumes than others.

In this case, thread 1 is completely saturated with wet volumes, followed by threads 2 and 3, while threads 4 to 8 are only assigned dry volumes. An alternative option is to completely interleave volumes and threads, as is shown in Fig. 15c, by using a small `b` value. This achieves a perfectly balanced load, through isotropic thread distribution in space, but the cycle iterations assigned to each thread become sparse and incur into cache pollution.

The best option is to interleave blocks that are large enough to saturate the cache memories on modern processors, while still providing a nearly isotropic distribution of threads throughout the domain (Fig. 15d). The optimal block size is such that it fits entirely in the cache memory, around 2MB per core on modern processors, holding approximately 12.5k sub-objects for this application with the SWE.

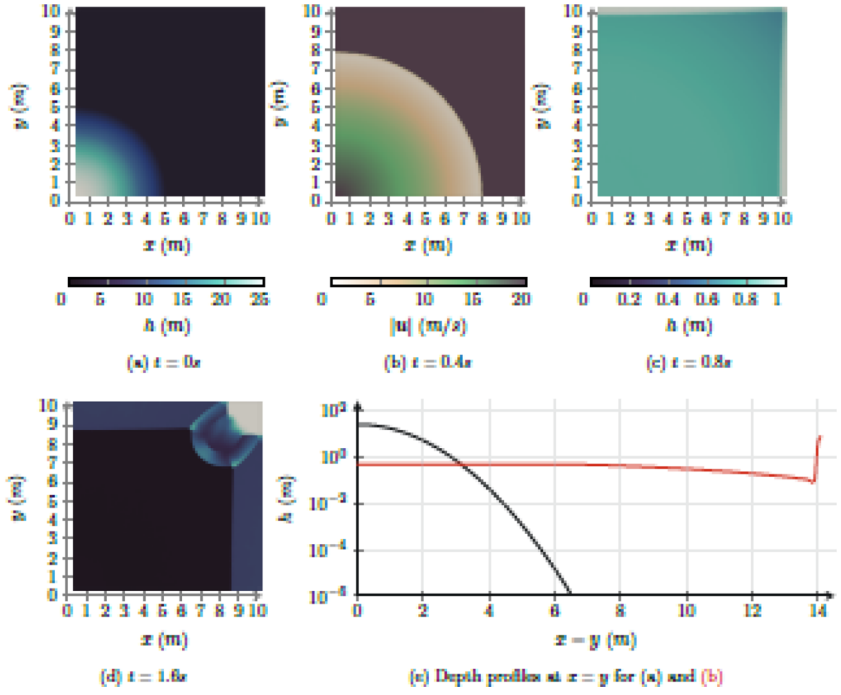
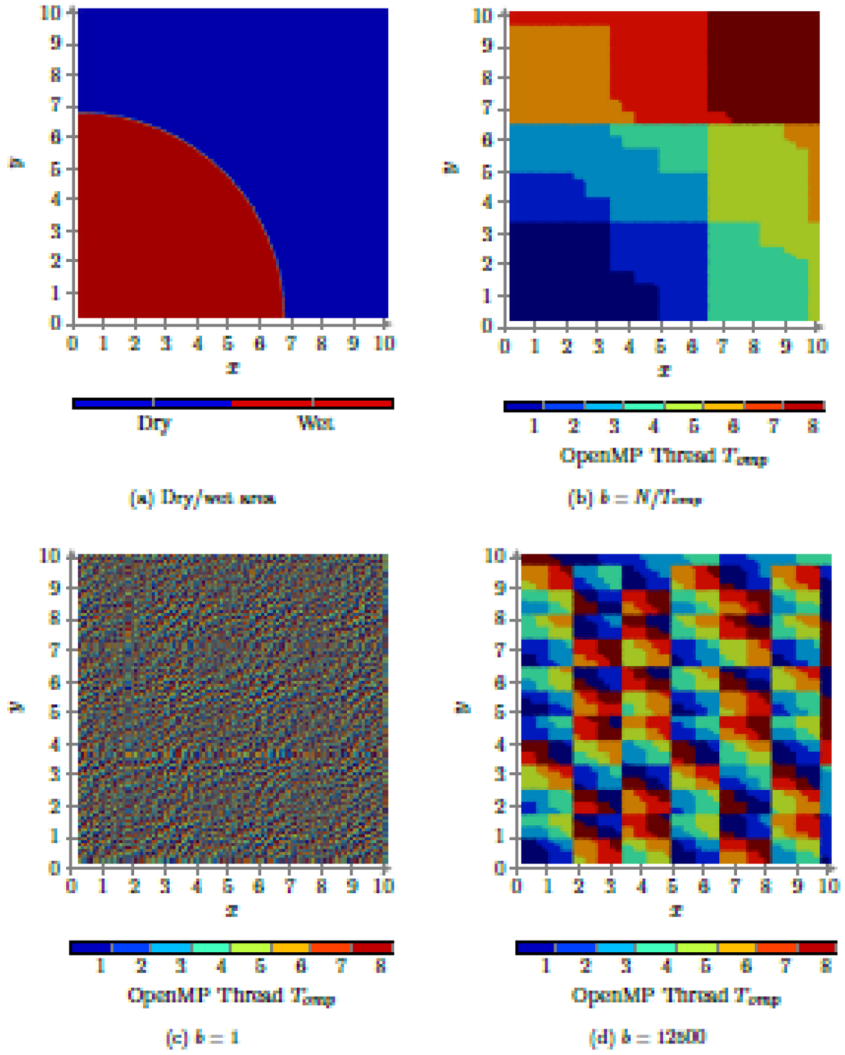


Fig. 12. Maximum OpenMP speedup for different code versions (left) and mesh sizes (right).



(a) Dry/wet areas (b) $b = N/T_{omp}$ (c) $b = 1$ (d) $b = 12800$

Fig. 13. Maximum speedup with CUDA (left) and effect of mesh size on GPU performance (right).

Fig. 16 (left) shows the speedup evolution, relative to the sequential execution, during the dynamic-load simulation for an 8 thread execution. In a contiguous domain division, $b = N/T_{omp}$, the speedup is initially much lower (2.67) than the one obtained for static loads (Fig. 12) since less than half of threads have wet cells. As the wave propagates, more threads start working and the throughput increases steadily up to that of a balanced simulation.

For the complete interleaving option, $b = 1$, the load is inherently balanced, with an almost constant speedup, although with a lower average value (3.8) due to the inefficient memory-access pattern. The contiguous block interleaving option obtains an increasing speedup with the largest time-averaged value (4.23), and a peak value comparable to the static load (4.35). This last option is the one adopted by default in Hi-STAV (as in Listing 1 and CPU code in Fig. 3).

The load balancing issue for GPUs is not as relevant, since CUDA only supports blocks of up to 1024 threads concurrently executed on

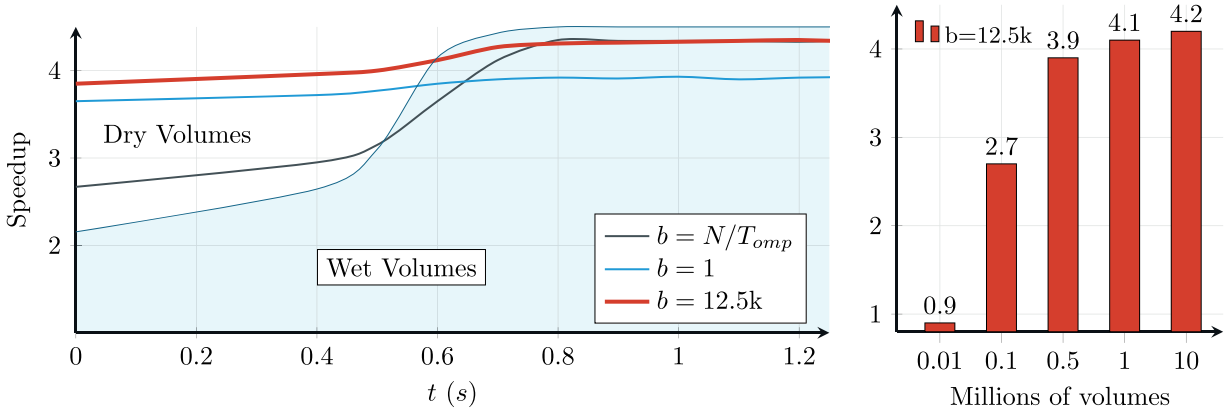
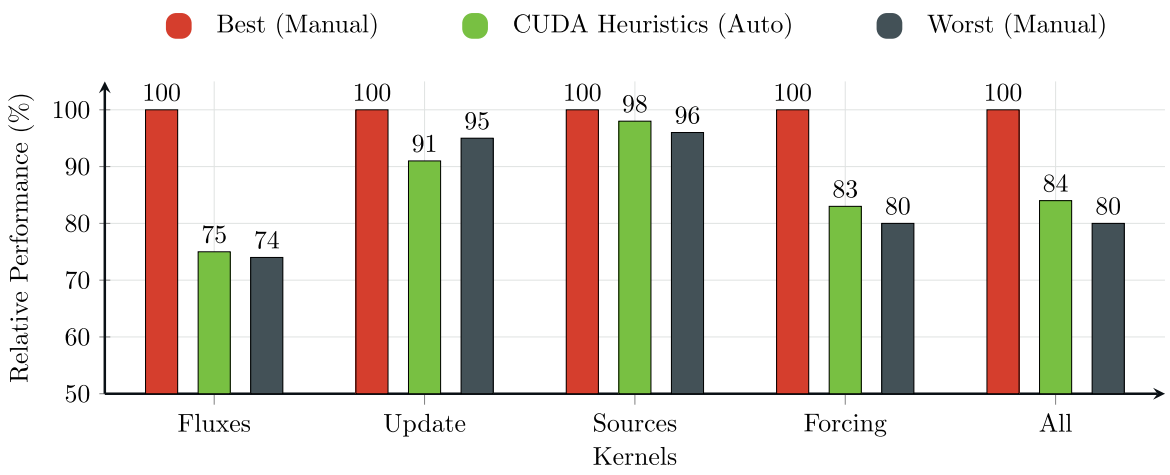


Fig. 14. Solution of the static-load test case.



	Kernel performance in MVol/s				
	Fluxes	Updates	Sources	Forcing	All
Best (M)	271	531	398	666	141
CUDA (A)	203	483	390	552	118
Worst (M)	200	505	382	532	112

Fig. 15. Distribution of threads throughout the domain.

```
#pragma omp parallel
{
  while (currentTime <= finalTime) {
    // ...
    // Flux computation
    #pragma omp for schedule(static, 12500)
    for (int i = 0; i < cpuMesh.numVols; ++i){
      cpuVol[i].flow->computeFluxes();
    // Time step reduction
    #pragma omp critical
    if (threadMinDt < cpuNumerics.dt)
      cpuNumerics.dt = threadMinDt;
    // Synchronization
    #pragma omp barrier
    // State update
    #pragma omp for schedule(static, 12500) nowait
    for (int i = 0; i < cpuMesh.numVols; ++i)
      cpuVol[i].flow->updateStates();
    // Apply sources
    #pragma omp for schedule(static, 12500) nowait
    for (int i = 0; i < cpuMesh.numVols; ++i)
      cpuVol[i].sources->applySources();
    // Apply forcing
    #pragma omp for schedule(static, 12500) nowait
    for (int i = 0; i < cpuMesh.numVols; ++i)
      cpuVol[i].forcing->applyForcing();
    // ...
  }
}
```

Fig. 16. Speedup evolution (left) and average speedup for different mesh sizes (right).

the device. This forcefully leads to an interleaved volume grouping, in a midterm of Fig. 15c and d.

4.4. Maximizing GPU performance

Each GPU has defined limits regarding the number of threads and blocks that can be used in kernels, and the choice of these parameters has a large impact on performance. Developers have to consider several factors in the tuning of these parameters but, in general terms, their aim is to achieve sufficient occupancy: the percentage of active threads is enough to hide memory and instruction-level latencies on the device [20]. The CUDA API provides the `cudaOccupancyMaxPotentialBlockSize` function, which heuristically targets an efficient thread-block configuration regarding occupancy. In our benchmarks with the `GTX 1080`, this tool was mostly reliable for simpler kernels, namely `applySources` and `applyUpdates`, whereas for more complex kernels, such as `computeFluxes`, the best results were obtained with manual profiling and tuning. The fact that more complex kernels feature more execution paths may turn occupancy as a misleading performance indicator, and partly explain the poor performance from the heuristics approach.

Fig. 17 shows the absolute and relative performances of the best, worst and heuristically calculated threads per block configuration for each kernel. The configurations provided by CUDA's API generally achieved a much lower throughput, with particular emphasis on the most resource-intensive `computeFluxes` kernel, where there was a performance drop of 25%. Overall, the heuristically calculated configurations provided a suboptimal performance, with a 16% performance penalty relative to the best achievable throughput.

5. Conclusions and future developments

Computational performance is often the limiting factor for various numerical modeling applications. Exploiting the growth of parallel processing hardware and coding tools, this work introduces and analyzes a unified framework for CPU + GPU explicit hyperbolic solvers.

The proposed solution, based on contextual object spliting (CoS), offers important advantages to developers working with heterogeneous applications: code reusability and maintainability, supported by an object-oriented approach; a unified source code, with no branching for CPU or GPU execution; a dissociation of physical and numerical features from the complexity associated to parallel environments; and a scalable performance in modern processors.

The different design optimizations herein presented and discussed

```
while (currentTime <= finalTime){
  // ...
  // Flux computation
  getFluxesKernel <<<fluxes.blocks, fluxes.threads>>> (gpuVol, numVols);
  // Time step reduction and CPU-GPU synchronization
  cpuNumerics.dt = reduceDtOnGPU();
  cudaMemcpyToSymbol(gpuNumerics.dt, &cpuNumerics.dt, sizeof(double));
  // State update
  updateStatesKernel <<<states.blocks, states.threads>>> (gpuVol, numVols);
  // Apply sources
  applySourcesKernel <<<sources.blocks, sources.threads>>> (gpuVol, numVols);
  // Apply forcing
  applyForcingKernel <<<forcing.blocks, forcing.threads>>> (gpuVol, numVols);
  //...
}
```

Fig. 17. Kernel performance on the GTX 1080.

require varying degrees of implementation efforts, namely in the case of already existent models, and were benchmarked individually. On CPUs, the fully-optimized code was capable of achieving a 149% performance improvement, relative to a naïve implementation, in single-threaded execution.

In terms of parallel scalability, the cumulative optimizations also led to supra-linear speedups on Hyper-Threading capable CPUs. As load-balancing becomes a central parameter for parallel performance, in the case of sparse simulations, a block-interleaving technique was proposed and shown to bound potential performance penalties to less than 10%.

Regarding GPU execution the expected speedups were much higher. The benchmark showed that the proposed design was able to achieve a very considerable speedup on modern hardware, nearly 40 times faster than sequential CPU execution, which greatly broadens the spectrum of model applicability. Specific parameters regarding GPU configuration were also analyzed, with considerable performance gains being obtained after manual profiling.

The following step in design development is to advance towards a fully *distributed* and *heterogeneous* execution, which constitutes a critical argument for designing unified CPU + GPU programs. The ability to include multiple devices of different architectures will contribute to an added computational capacity, unlocking larger application scales at increased spatial and physical details.

CRediT authorship contribution statement

Daniel A.S. Conde: Conceptualization, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Ricardo B. Canelas:** Formal analysis, Investigation, Writing - review & editing. **Rui M.L. Ferreira:** Conceptualization, Formal analysis, Funding acquisition, Project administration, Resources, Supervision, Writing - review & editing.

Declaration of Competing Interest

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

Acknowledgments

This research was partially supported by Portuguese and European funds, within the COMPETE 2020 and PORL-FEDER programs, through project RiverCure PTDC/CTA-OHR/29360/2017.

Appendix. Discretization of the shallow-water equations

Recalling the Gudonov-type finite volume formulation from Eq. (4):

$$\frac{\boldsymbol{U}_i^{n+1} - \boldsymbol{U}_i^n}{\Delta t} A_i + \sum_{k=1}^{K_i} \left(\underset{k}{\Delta} \boldsymbol{E}_i^n - \boldsymbol{T}_{i_k}^n \right) \boldsymbol{n}_{i_k} l_{i_k} = \boldsymbol{S}_i^n A_i \tag{11}$$

the jump in the projected conservative fluxes across each edge can be locally linearized by

$$\underset{k}{\Delta} \boldsymbol{E}_i \boldsymbol{n}_{i_k} = \tilde{\boldsymbol{J}}_{\boldsymbol{i_k}} \underset{k}{\Delta} \boldsymbol{U}_i = \tilde{\boldsymbol{J}}_{\boldsymbol{i_k}} \left(\boldsymbol{U}_{j(i_k)} - \boldsymbol{U}_i \right) \tag{12}$$

where $\tilde{\boldsymbol{J}}_{\boldsymbol{i_k}}$ is an approximate Jacobian of the conservative fluxes $\boldsymbol{E}_{i_k}$, as proposed by Roe [25]. The Jacobian is diagonalizable with M linearly independent right-eigenvectors $\tilde{\boldsymbol{R}}_{\boldsymbol{i_k}}$ and M real eigenvalues $\tilde{\boldsymbol{\Lambda}}_{\boldsymbol{i_k}}$, in the form

$$\tilde{\boldsymbol{J}}_{i_k} = \tilde{\boldsymbol{R}}_{i_k} \tilde{\boldsymbol{\Lambda}}_{i_k} \tilde{\boldsymbol{R}}_{i_k}^{-1} \tag{13}$$

where the eigenvalues $\tilde{\boldsymbol{\Lambda}}$ are further split as $\tilde{\boldsymbol{\Lambda}} = \tilde{\boldsymbol{\Lambda}}^- + \tilde{\boldsymbol{\Lambda}}^+$, so that $\tilde{\lambda}_m^- \leq 0$ and $\tilde{\lambda}_m^+ \geq 0$, respectively holding the inward (-) or outward (+) going fluxes. The eigenbasis formed from $\tilde{\boldsymbol{R}}$ is used to project both $\underset{k}{\Delta} \boldsymbol{U}_i$ and $\boldsymbol{T}_{i_k}$, with the respective wave-strengths α and source-strengths β being algebraically solved from

$$\underset{k}{\Delta} \boldsymbol{U}_i = \tilde{\boldsymbol{R}}_{\boldsymbol{i_k}} \boldsymbol{A}_{\boldsymbol{i_k}} = \tilde{\boldsymbol{R}}_{\boldsymbol{i_k}} (\alpha_1 \ \cdots \ \alpha_M)_{i_k}^T \tag{14}$$

$$(\boldsymbol{T}\boldsymbol{n})_{i_k} = \widetilde{\boldsymbol{R}}_{i_k} \widetilde{\boldsymbol{B}}_{i_k} = \widetilde{\boldsymbol{R}}_{\boldsymbol{i_k}} (\beta_1 \cdots \beta_M)_{i_k}^T \tag{15}$$

where M is the total number of state variables and conservation laws. Substituting Eqs. (12) to (15) in Eq. (4), and considering only the incoming fluxes $\tilde{\boldsymbol{\Lambda}}^-$ when re-averaging the solution, gives

$$\frac{\boldsymbol{U}_i^{n+1}-\boldsymbol{U}_i^n}{\Delta t}A_i+\sum_{k=1}^{K_i}\tilde{\boldsymbol{R}}_{i_k}^n(\bar{\boldsymbol{\Lambda}}\bar{\boldsymbol{A}}-\boldsymbol{B})_{i_k}^n l_{i_k}=\boldsymbol{S}_i^n A_i$$

which can be expanded and rearranged in order to the updated state variables $\boldsymbol{U}_i^{n+1}$ as

$$\boldsymbol{U}_i^{n+1}=\boldsymbol{U}_i^n-\frac{\Delta t}{A_i}\sum_{k=1}^{K_i}\sum_{m=1}^{M}\tilde{\boldsymbol{e}}_{m,i_k}^n(\bar{\tilde{\lambda}}_m\bar{\alpha}_m-\beta_m)_{i_k}^n l_{i_k}+\Delta t\,\boldsymbol{S}_i^n \tag{16}$$

where $\tilde{\lambda}_m$ and $\tilde{\boldsymbol{e}}_m$ are the M eigenvalues and corresponding eigenvectors of $\tilde{\boldsymbol{J}}_{i_k}$. For the shallow-water equations with scalar transport, where

$$\boldsymbol{U}=\begin{pmatrix}h\\hu\\hv\\h\zeta_1\\\vdots\\h\zeta_S\end{pmatrix}\quad \boldsymbol{E}=\begin{pmatrix}hu & hv\\ hu^2+gh^2/2 & hvu\\ huv & hv^2+gh^2/2\\ hu\zeta_1 & hv\zeta_1\\ \vdots & \vdots\\ hu\zeta_S & hv\zeta_S\end{pmatrix}\quad \boldsymbol{T}=-gh\begin{pmatrix}0&0\\ \Delta z_b&0\\0&\Delta z_b\\0&0\\\vdots&\vdots\\0&0\end{pmatrix}\quad \boldsymbol{S}=\begin{pmatrix}i_p-\partial_t Z_b\\(\tau_w-\tau_b)_x/\rho\\(\tau_w-\tau_b)_y/\rho\\\phi_1\\\vdots\\\phi_s\end{pmatrix}$$

the following eigen-structure is obtained

$$\tilde{\boldsymbol{\Lambda}}_{i_k}=\begin{pmatrix}\tilde{\boldsymbol{u}}\boldsymbol{n}-\tilde{c}&0&0&0&\cdots&0\\0&\tilde{\boldsymbol{u}}\boldsymbol{n}&0&\vdots&\ddots&\vdots\\0&0&\tilde{\boldsymbol{u}}\boldsymbol{n}+\tilde{c}&0&\cdots&0\\0&\cdots&0&\tilde{\boldsymbol{u}}\boldsymbol{n}&\cdots&0\\\vdots&\ddots&\vdots&\vdots&\ddots&\vdots\\0&\cdots&0&0&\cdots&\tilde{\boldsymbol{u}}\boldsymbol{n}\end{pmatrix}_{i_k}\quad \tilde{\boldsymbol{R}}_{i_k}=\begin{pmatrix}1&0&1&0&\cdots&0\\\tilde{u}-\tilde{c}n_x&-\tilde{c}n_y&\tilde{u}+\tilde{c}n_x&\vdots&\ddots&\vdots\\\tilde{v}-\tilde{c}n_y&\tilde{c}n_x&\tilde{v}+\tilde{c}n_y&0&\cdots&0\\\tilde{\zeta}_1&0&\tilde{\zeta}_1&1&\cdots&0\\\vdots&\vdots&\vdots&\vdots&\ddots&\vdots\\\tilde{\zeta}_S&0&\tilde{\zeta}_S&0&\cdots&1\end{pmatrix}_{i_k}$$

where the tilde denotes Roe-averaged variables [19,25] and c is the shallow wave celerity. Considering $\boldsymbol{t}_{i_k}=(-n_y,\,n_x)_{i_k}$ as the tangent to edge i_k, the wave and source-strengths in (14) and (15) are

$$\boldsymbol{A}_{i_k}=\begin{pmatrix}\frac{\Delta}{k}\tilde{h}/2+\frac{1}{2\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{n}_{i_k}\\ \frac{1}{\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{t}_{i_k}\\ \frac{\Delta}{k}\tilde{h}/2-\frac{1}{2\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{n}_{i_k}\\ \frac{\Delta}{k}h\zeta_1-\tilde{\zeta}_1\frac{\Delta}{k}h\\\vdots\\ \frac{\Delta}{k}h\zeta_S-\tilde{\zeta}_S\frac{\Delta}{k}h\end{pmatrix}_{i_k}\quad \boldsymbol{B}_{i_k}=\frac{\tilde{c}}{2}\begin{pmatrix}-\Delta z_b\\0\\\Delta z_b\\0\\\vdots\\0\end{pmatrix}_{i_k}$$

thus completing the necessary definitions for Eq. (16) regarding the shallow-water system.

The previous system has a computational complexity of $O((3+S)^2)$, meaning that model performance is mostly defined by the number of transported scalars. Attending to the patterns in the eigen-structure and respective projections, the complexity of the system can be reduced to $O(3^2+S)$, if the hydrodynamics and scalar transports are computed sequentially. Defining the hydrodynamics component of the system as the following η-indexed terms

$$\boldsymbol{U}_{\eta,i}=\begin{pmatrix}h\\hu\\hv\end{pmatrix}_i,\quad \tilde{\boldsymbol{\Lambda}}_{\eta,i_k}=\begin{pmatrix}\tilde{\boldsymbol{u}}\boldsymbol{n}-\tilde{c}&0&0\\0&\tilde{\boldsymbol{u}}\boldsymbol{n}&0\\0&0&\tilde{\boldsymbol{u}}\boldsymbol{n}+\tilde{c}\end{pmatrix}_{i_k}\quad \tilde{\boldsymbol{R}}_{\eta,i_k}=\begin{pmatrix}1&0&1\\\tilde{u}-\tilde{c}n_x&-\tilde{c}n_y&\tilde{u}+\tilde{c}n_x\\\tilde{v}-\tilde{c}n_y&\tilde{c}n_x&\tilde{v}+\tilde{c}n_y\end{pmatrix}_{i_k}$$

$$\boldsymbol{A}_{\eta,i_k}=\begin{pmatrix}\frac{\Delta}{k}\tilde{h}/2+\frac{1}{2\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{n}_{i_k}\\ \frac{1}{\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{t}_{i_k}\\ \frac{\Delta}{k}\tilde{h}/2-\frac{1}{2\tilde{c}}\left(\frac{\Delta}{k}h\boldsymbol{u}-\tilde{\boldsymbol{u}}\frac{\Delta}{k}h\right)\cdot\boldsymbol{n}_{i_k}\end{pmatrix}_{\eta,i_k}\quad \boldsymbol{B}_{\eta,i_k}=\frac{\tilde{c}}{2}\begin{pmatrix}-\Delta z_b\\0\\\Delta z_b\end{pmatrix}_{\eta,i_k}\quad \boldsymbol{S}_{\eta,i}=\begin{pmatrix}i_p-\partial_t Z_b\\(\tau_w-\tau_b)_x/\rho\\(\tau_w-\tau_b)_y/\rho\end{pmatrix}_i$$

Eq. (16) can be split into hydrodynamics and scalar updates, respectively given by (17) and (18)

$$\boldsymbol{U}_{\eta,i}^{n+1}=\boldsymbol{U}_{\eta,i}^n-\frac{\Delta t}{A_i}\sum_{k=1}^{K_i}\sum_{m=1}^{3}(\tilde{\boldsymbol{e}}_m^n)_{\eta,i_k}(\bar{\tilde{\lambda}}_m\bar{\alpha}_m-\beta_m)_{\eta,i_k}^n l_{i_k}+\Delta t\,\boldsymbol{S}_{\eta,i}^n \tag{17}$$

$$(h\zeta_s)_i^{n+1}=(h\zeta_s)_i^n-\frac{\Delta t}{A_i}\sum_{k=1}^{K_i}\left(\tilde{\zeta}_s\left(f_m^-\frac{\Delta}{k}h\right)+\bar{\tilde{\lambda}}_2\frac{\Delta}{k}(h\zeta_s)\right)_{i_k}^n l_{i_k}+\Delta t\phi_{s,i}^n \tag{18}$$

where $f_m^- = (\tilde{\lambda}_1^- \alpha_1 + \tilde{\lambda}_3^- \alpha_3)$ represents the inward conservative mass fluxes.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.advengsoft.2020.102802 .

References

[1] Asanovic K, Wawrzynek J, Wessel D, Yelick K, Bodik R, Demmel J, et al. A view of the parallel computing landscape. Communications of the ACM 2009;52(10):56. https://doi.org/10.1145/1562764.1562783. http://portal.acm.org/citation.cfm?doid=1562764.1562783

[2] Bryant R, O'Hallaron D. Computer systems: a programmers perspective. 3rd ed. PrenticeHall; 2015.

[3] Canelas R, Murillo J, Ferreira RM. Two-dimensional depth-averaged modelling of dam-break flows over mobile beds. Journal of Hydraulic Research 2013;51(4):392–407. https://doi.org/10.1080/00221686.2013.798891. http://www.tandfonline.com/doi/abs/10.1080/00221686.2013.798891

[4] Chen HL, Chang YI. Neighbor-finding based on space-filling curves. Inf Syst 2005;30(3):205–26. https://doi.org/10.1016/j.is.2003.12.002.

[5] Chilimbi TM, Hill MD, Larus JR. Cache-conscious structure layout. Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation - PLDI '99. 1. 1999. p. 1–12. https://doi.org/10.1145/301618.301633. ISBN 1581130945; http://portal.acm.org/citation.cfm?doid=301618.301633

[6] Conde DAS, Baptista MAV, Sousa Oliveira C, Ferreira RML. A shallow-flow model for the propagation of tsunamis over complex geometries and mobile beds. Natural Hazards and Earth System Science 2013;13(10):2533–42. https://doi.org/10.5194/nhess-13-2533-2013. http://www.nat-hazards-earth-syst-sci.net/13/2533/2013/

[7] Conde DAS, Telhado MJ, Viana Baptista MA, Ferreira RML. Severity and exposure associated with tsunami actions in urban waterfronts: the case of Lisbon, Portugal. Nat Haz 2015;79(3):2125–44.

[8] Dawes W, Harvey S, Fellows S. A practical demonstration of scalable, parallel mesh generation. 47th AIAA Aerospace sciences meeting & exhibit, 5-8 January 2009 Orlando FL. 1. 2009. p. 5–8. https://doi.org/10.2514/6.2009-981. http://arc.aiaa.org/doi/pdf/10.2514/6.2009-981

[9] de la Asunción M, Castro MJ, Fernández-Nieto ED, Mantas JM, Acosta SO, González-Vida JM. Efficient GPU implementation of a two waves TVD-WAF method for the two-dimensional one layer shallow water system on structured meshes. Comput Fluids 2013;80(1):441–52. https://doi.org/10.1016/j.compfluid.2012.01.012.

[10] Ferreira RML, Franca MJ, Leal JGaB, Cardoso AH. Mathematical modelling of shallow flows: closure models drawn from grain-scale mechanics of sediment transport and flow hydrodynamics. Can J Civ Eng 2009;36(2009):1605–21. https://doi.org/10.1139/L09-033.

[11] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns - Elements of Reusable Object-Oriented Software. 2002. https://doi.org/10.1093/carcin/bgs084. ISBN 9780201715941. http://books.google.com/books?id=JPOaP7cyk6wC{&}pg=PA78{&}dq=intitle:Design+Patterns+Elements+of+Reusable+Object+Oriented+Software{&}hl={&}cd=3{&}source=gbs{_}api{%}5Cnpapers2://publication/uuid/944613AA-7124-44A4-B86F-C7B2123344F3

[12] Lacasta A, Morales-Hernández M, Murillo J, García-Navarro P. An optimized GPU implementation of a 2D free surface simulation model on unstructured meshes. Adv Eng Softw 2014;78:1–15. https://doi.org/10.1016/j.advengsoft.2014.08.007.

[13] LeFloch PG. Hyperbolic systems of conservation laws: the theory of classical and nonclassical shock waves. Birkhäuser Basel; 2002.

[14] Leveque RJ, George DL. High-resolution finite volume methods for the shallow water equations with bathymetry and dry states. Adv Coast Ocean Eng 2008;10:43–73. https://doi.org/10.1142/9789812790910_0002.

[15] Liu JY, Smith MR, Kuo FA, Wu JS. Hybrid OpenMP/AVX acceleration of a split HLL finite volume method for the shallow water and Euler equations. Comput Fluids 2015;110:181–8. https://doi.org/10.1016/j.compfluid.2014.11.011.

[16] Macías J, Mercado A, González-Vida JM, Ortega S, Castro MJ. Comparison and computational performance of tsunami-HySEA and MOST models for LANTEX 2013 scenario: impact assessment on puerto rico coasts. Pure Appl Geophys 2016;173(12):3973–97. https://doi.org/10.1007/s00024-016-1387-8.

[17] Murillo J, García-Navarro P. Weak solutions for partial differential equations with source terms: application to the shallow water equations. J Comput Phys 2010;229(11):4327–68. https://doi.org/10.1016/j.jcp.2010.02.016.

[18] Murillo J, García-Navarro P, Burguete J. Conservative numerical simulation of multi-component transport in two-dimensional unsteady shallow water flow. J Comput Phys 2009;228(15):5539–73. https://doi.org/10.1016/j.jcp.2009.04.039.

[19] Murillo J, Latorre B, García-Navarro P. A Riemann solver for unsteady computation of 2D shallow flows with variable density. J Comput Phys 2012;231(14):4775–807. https://doi.org/10.1016/j.jcp.2012.03.016.

[20] NVidia. Cuda C programming guide. nVidia Programming Guides. PG-02829-0. 2017. p. 1–301.

[21] OpenMP. OpenMP 4.5 application programming interface. Tech. Rep.. Open Multi-Processing; 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[22] Parés C, Castro M. On the well-balance property of Roe's method for non-conservative hyperbolic systems. applications to shallow-water systems. ESAIM: Mathematical Modelling and Numerical Analysis 2004;38(5):821–52. https://doi.org/10.1051/m2an:2004041. http://www.esaim-m2an.org/10.1051/m2an:2004041

[23] PassMark. CPU and GPU benchmarks. 2018. https://www.passmark.com.

[24] Reguly IZ, Giles D, Gopinathan D, Quivy L, Beck JH, Giles MB, et al. The VOLNA-OP2 tsunami code (version 1.5). Geosci Model Dev 2018;11(11):4621–35. https://doi.org/10.5194/gmd-11-4621-2018.

[25] Roe PL. Approximate Riemann solvers, parameter vectors, and difference schemes. J Comput Phys 1981;43(2):357–72. https://doi.org/10.1016/0021-9991(81)90128-5.

[26] Sung IJ, Liu GD, Hwu WMW. DL: a data layout transformation system for heterogeneous computing. 2012 Innovative parallel computing, InPar 2012. 2012. https://doi.org/10.1109/InPar.2012.6339606.

[27] Toro EF, Vázquez-Cendón ME. Flux splitting schemes for the Euler equations. Comput Fluids 2012;70:1–12. https://doi.org/10.1016/j.compfluid.2012.08.023.

[28] Vacondio R, Dal Palú A, Mignosa P. GPU-enhanced finite volume shallow water solver for fast flood simulations. Environ Modell Software 2014;57:60–75. https://doi.org/10.1016/j.envsoft.2014.02.003.

[29] Vetter JS. Contemporary High Performance Computing: From Petascale Toward Exascale. 2. CRC Press; 2015. ISBN 978-1-46-656835-8. http://www.routledge.com/books/details/9781466568341/

[30] Zhang S, Xia Z, Yuan R, Jiang X. Parallel computation of a dam-break flow model using OpenMP on a multi-core computer. J Hydrol 2014;512:126–33. https://doi.org/10.1016/j.jhydrol.2014.02.035.