Environmental Modelling & Software 57 (2014) 60–75

Contents lists available at ScienceDirect

Environmental Modelling & Software

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

GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations

R. Vacondio [a,*], A. Dal Palù [b], P. Mignosa [a]

[a] DICATeA, University of Parma, v.le Parco Area delle Scienze 181/A, 43124 Parma, Italy
[b] Department of Mathematics and Computer Science, University of Parma, v.le Parco Area delle Scienze 53/A, 43124 Parma, Italy

ARTICLE INFO

Article history:
Received 13 June 2013
Received in revised form
3 February 2014
Accepted 4 February 2014
Available online 6 March 2014

Keywords:
Flood simulation
Parallel computing
GPU
Shallow Water
Finite Volume

ABSTRACT

In this paper a parallelization of a Shallow Water numerical scheme suitable for Graphics Processor Unit (GPU) architectures under the NVIDIA™'s Compute Unified Device Architecture (CUDA) framework is presented. In order to provide robust and accurate simulations of real flood events, the system features a state-of-the-art Finite Volume explicit discretization technique which is well balanced, second order accurate and based on positive depth reconstruction. The model is based on a Cartesian grid and boundary conditions are implemented by means of the implicit local ghost cell approach, which enables the discretization of a broad spectrum of boundary conditions including inflow/outflow conditions. A novel and efficient Block Deactivation Optimization procedure has also been adopted, in order to increase the efficiency of the numerical scheme in the presence of wetting-drying fronts. This led to speedups of two orders of magnitude with respect to a single-core CPU. The code has been validated against several severe benchmark test cases, and its capability of producing accurate fast simulations (with high ratios between physical and computing times) for different real world cases has been shown.

Software availability

Name of software: G-Flood
Contact address: DICATeA, University of Parma, v.le Parco Area delle Scienze 181/A, 43124 Parma, Italy
Email: renato.vacondio@unipr.it, alessadro.dalpalu@unipr.it
Language: CUDA, C++
Hardware: CUDA- enabled GPU
Availability: upon request by email (for scientific collaboration)
Year first available: 2014

1. Introduction

Nowadays, the flood risk assessment has been shown to be of key importance, in order to minimize damages and economic losses caused by floods. The ClimateCost project (Feyen and Watkiss, 2011) estimates that, without any modification of the available flood defense systems, the expected damages in 2050 will be worth 46 billion Euro/year (for the 27 countries of the European Union) with about 170,000 people being involved each year. It is therefore straightforward that the flood risk assessment represents a significant challenge, also from an economical point of view. Several international agencies (Belmont Forum International Group of Funding Agencies for Global Change, European Union Directive 2007/60/EC on the assessment and management of flood risks) have recently suggested that the reduction of the hydraulic risk, obtained by maximizing the resilience (defined as the capability to prepare, modify, anticipate and recover), might be more effective than building structural defense systems (such as levees).

However, the first approach, while being advantageous both from the economic and political point of view, requires the extensive use of fast and accurate mathematical modeling. Ahead of impending flood events, the numerical simulation might produce real-time forecast of flooded areas and thus might be used to minimize the disruptive effect of the event. Flood modeling is also necessary to define the flood-hazard maps for a given return period. Modern probabilistic approaches (mainly based on Montercarlo-like methods), allow to evaluate the uncertainties of the parameters considered in the hydraulic simulation, but also require the simulation of many different scenarios and thus are computationally expensive.

* Corresponding author.
E-mail addresses: renato.vacondio@unipr.it, rvacondio@gmail.com (R. Vacondio), alessandro.dalpalu@unipr.it (A. Dal Palù), paolo.mignosa@unipr.it (P. Mignosa).

http://dx.doi.org/10.1016/j.envsoft.2014.02.003
1364-8152/

Two dimensional Shallow Water Equations (SWEs) are used to simulate a broad variety of free surface flows, such as dam-breaks, river floods, flood plain inundations, etc. and they have been discretized by different numerical methods (Casulli, 1990; Miglio et al., 1999; Vacondio et al., 2012; Bates and Roo, 2000). Explicit Finite Volume (FV) schemes, based on Riemann solvers, are able to accurately reproduce supercritical, subcritical and transient flows, including shock-type flow discontinuities (Toro, 1999a,b; LeVeque, 2002). However, flooding simulation over irregular bathymetries requires some specific features, such as (a) a robust treatment of wetting and drying fronts (Balzano, 1998; Bates and Hervouet, 1999; Defina, 2000; Bates and Horritt, 2005; Begnudelli and Sanders, 2006; Liang and Borthwick, 2009; Bradford and Sanders, 2002), (b) the imposition of open boundary conditions, (c) an accurate discretization of slope and friction source terms (Toro, 1999a,b; Bermúdez and Vázquez, 1994; García-Navarro and Vázquez, 2000; Bermúdez et al., 1998; Vázquez, 1999; Hubbard and García-Navarro, 2000; LeVeque, 1998; Zhou et al., 2001; Valiani and Begnudelli, 2006a,b; Soares-Frazo et al., 2007) possibly with (d) a well-balanced (or C-property) approach leading to the capability to preserve steady state at rest (Bermúdez and Vázquez, 1994; Greenberg and Leroux, 1996; Rogers et al., 2003; Liang and Marche, 2009).

FV Explicit schemes can provide all the above-mentioned properties, but their use for high resolution simulations of large domains is prevented by their high computational cost: flood inundations at large scale are often simulated with 2D-simplified kinematic wave approximation of SWEs (Bates and Roo, 2000). Alternatively, if the fully 2D Shallow Water Equations are solved, coarse grid is necessarily adopted in order to keep the simulation size within the limits of the hardware capabilities. Both these approaches aim at reducing the total computational time, even if simplified models are not always effective, as shown by Neal et al. (2012); Néelz and Pender (2010), but they obviously decrease the overall accuracy of the simulation. To address this issue, Sanders et al. (2010) developed a parallel FV explicit solver, built over the Message Passing Interface (MPI), which can be run on large cluster machines. This solver can simulate flood inundation at regional scale by using millions of cells and high resolution unstructured mesh. Neal et al. (2010) compared different parallelization methods. Recently Akbar and Aliabadi (2013) have developed an implicit numerical scheme which can use a large timestep and can be used to simulate over large scale inundations.

In this work we take advantage of a relatively recent capability offered by dedicated Graphics Cards (Graphics Processing Unit, in short GPU), which can be used to perform High Performance Computing on classical personal computers. When compared to traditional clusters and supercomputers, the striking difference offered by GPUs is the low (and fast decreasing) cost per processor and the fact that thousands of parallel tasks can be performed in parallel on the same card. The GPUs were originally designed for rendering complex 3D scenes, which intrinsically involve a high degree of parallel computations. The particular feature of these operations is that the same instructions (e.g. matrix multiplications) can be performed in parallel over different data. This particular type of parallelism, named Single Instruction Multiple Thread (SIMT), has been implemented on the GPUs with dedicated hardware. A GPU contains a large set of Arithmetic Logic Units (ALUs) (that can perform their tasks concurrently) controlled by some Control Units that apply the same instruction in parallel to different operands on the corresponding ALUs. Only recently, the video cards allowed a General Purpose programming of the hardware (therefore the acronym GPGPU), making it possible to exploit the multicore architecture for user defined tasks, rather than graphical rendering and processing of images. The main competitor NVIDIA™ released a framework named CUDA that allows to program the GPU with conventional programming languages. The interested reader is referred to cud (Cuda best practices guide); Brodtkorb et al. (2013) for details about GPU programming and best practices.

Recently, GPUs have been successfully exploited to accelerate numerical simulations in different scientific fields, such as computational biophysics (Owens et al., 2008), molecular dynamics, medical imaging, seismic imaging and fluid dynamics (Garland et al., 2008; Crespo et al., 2011). There were previous proposals to implement FV Shallow Water numerical schemes on GPUs. A first order accurate FV numerical scheme has been ported to GPUs by Lastra et al. (2009) using OpenGL and Cg shading language (Fernando and Kilgard, 2003). Neal et al. (2010) presented a comparison between different parallelization techniques, including a Single Instruction Multiple Data technique adopting ClearSpeed acceleration cards. Subsequently a Roe scheme was presented by de la Asunción et al. (2011), a first order accurate, finite difference model is adopted by Kalyanapu et al. (2011) and Lamb et al. (2009) developed a diffusion wave model, based on CUDA programming language. More recently, several attempts to develop numerical schemes with high order of accuracy have been proposed (Gallardo et al., 2011; de la Asunción et al., 2013) also adopting unstructured mesh (Castro et al., 2011; de la Asunción et al., 2013). Simulations of practical test cases have been reported by de la Asunción et al. (2013); Brodtkorb et al. (2012) which showed relevant speed-ups (in comparison with non-parallel codes) for dam-break waves. Moreover, attempts to use multiple GPUs to tackle larger simulations have been presented (Saetra and Brodtkorb, 2012).

In this paper, an efficient CUDA implementation of a first and second order FV numerical scheme on a Cartesian grid with the positivity-preserving hydrostatic reconstruction proposed by Audusse et al. (2004) is presented. In order to guarantee the C-property, the pre-balanced version of the SWEs (Rogers et al., 2003; Liang and Marche, 2009) has been used.

In this way, the numerical scheme ensures a robust treatment of wet-dry fronts and allows to accurately simulate floodings over real irregular bathymetries. Moreover, the algorithm can handle open boundaries through an implicit local ghost cell approach (refer to Section 3.4 for more details) and, unlike the global approach (Brodtkorb et al., 2012), it enables the imposition of mixed boundary conditions for a given cell. The implicit local ghost approach is slightly more computationally expensive than the global one, but it is necessary to simulate real-world cases where mixed open and closed boundary conditions are often present.

To remarkably increase the efficiency of the numerical scheme in presence of partially dry bathymetries, a novel and efficient Block Deactivation Optimization (BDO) procedure has been developed. This technique is based on the block partitioning of the simulation area and on the identification of blocks that are expected to contain wet cells. This allows to reduce the number of cells to be considered, while maintaining the correctness of the simulation.

The paper is organized as follows: in Section 2 the explicit Finite Volume numerical scheme for the Shallow Water Equations (SWEs) is briefly described. In Section 3 the details of the CUDA implementation, including the different optimization strategies adopted, are reported. In Section 4 the accuracy of the numerical model is verified against different challenging test cases, whereas the numerical scheme capability to efficiently simulate real floodings is assessed by reproducing two different real-life test cases.

2. Numerical model

$$\frac{\partial}{\partial t}\int_A \mathbf{U}\mathrm{d}A + \int_C \mathbf{H}\cdot\mathbf{n}\mathrm{d}C = \int_A \left(\mathbf{S}_0 + \mathbf{S}_f\right)\mathrm{d}A. \quad (1)$$

In Equation (1) A is the area of the integration element, C the element boundary, $\mathbf{n}$ the outward unit vector normal to C, $\mathbf{U}$ the vector of the conserved variables and $\mathbf{H} = (\mathbf{F}, \mathbf{G})$ the tensor of fluxes in the x and y directions:

$$\mathbf{U} = \begin{bmatrix} h \\ uh \\ vh \end{bmatrix}, \ \mathbf{F} = \begin{bmatrix} uh \\ u^2h + \frac{1}{2}gh^2 \\ uvh \end{bmatrix}, \ \mathbf{G} = \begin{bmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}gh^2 \end{bmatrix}, \quad (2)$$

where h is the flow depth, u and v are the velocity components in the x and y directions and g is the acceleration due to gravity. The bed and friction slope source terms $\mathbf{S}_0$ and $\mathbf{S}_f$ are expressed according to the following relations:

$$\mathbf{S}_0 = \left[0;\ -gh\frac{\partial z}{\partial x};\ -gh\frac{\partial z}{\partial y}\right]^T, \ \mathbf{S}_f = \left[0;\ -gh\frac{n_f^2 u\sqrt{u^2+v^2}}{h^{4/3}};\ -gh\frac{n_f^2 v\sqrt{u^2+v^2}}{h^{4/3}}\right]^T, \quad (3)$$

in which z is the bed elevation with respect to a horizontal reference plane and n_f is the roughness coefficient according to the Manning equation. The numerical discretization of the bed slope source term is described in Appendix B. For the friction source term the implicit discretization of Caleffi et al. (2003) is adopted: this prevents the formation of spurious oscillations in the presence of very small depths, which might occur when a naive explicit formulation is used.

The problem of obtaining a balance between fluxes and source terms in Equation (1) has received some attention in the recent literature. Some authors addressed the problem rectifying the SWEs formulation by numerical treatment (Zhou et al., 2001; Aureli et al., 2008; Valiani and Begnudelli, 2006a,b). More recently, Rogers et al. (2003) derived an algebraic modification of the Equation (1) able to balance the hyperbolic system of equations, regardless of the adopted numerical scheme. Liang and Borthwick (2009) further modified the idea presented by Rogers et al. (2003), by proposing a different way to manipulate the SWEs, which is suitable also for problems with wet-dry interfaces. In this formulation, herein adopted, the terms of Equation (2) are modified as follows:

$$\mathbf{U} = \begin{bmatrix} \eta \\ uh \\ vh \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} uh \\ u^2h + \frac{1}{2}g(\eta^2 - 2\eta z) \\ uvh \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}(g\eta^2 - 2\eta z) \end{bmatrix}, \quad (4)$$

where $\eta = h + z$ is the free surface elevation above datum and the bottom source term $\mathbf{S}_0$ is:

$$\mathbf{S}_0 = \left[0;\ -g\eta\frac{\partial z}{\partial x};\ -g\eta\frac{\partial z}{\partial y}\right]^T. \quad (5)$$

The partial differential Equation (1) is solved on a Cartesian grid. Both a first order and a second order accurate (in space and time) Finite Volume (FV) numerical approximation of the SWEs have been implemented.

Regardless of the order of accuracy, the fluxes are calculated using a HLLC approximate Riemann solver (Toro, 1999a,b).

For the first order approximation, the vector of the conserved physical quantities $\mathbf{U}_{i,j}$ is updated in time as follows (Toro, 1999a,b):

$$\mathbf{U}_{i,j}^{n+1} = \mathbf{U}_{i,j}^{n} - \frac{\Delta t^n}{\Delta x}\left(\mathbf{F}_{i+\frac{1}{2},j} - \mathbf{F}_{i-\frac{1}{2},j}\right) - \frac{\Delta t^n}{\Delta y}\left(\mathbf{G}_{i,j+\frac{1}{2}} - \mathbf{G}_{i,j-\frac{1}{2}}\right) + \Delta t^n\left(\mathbf{S}_0 + \mathbf{S}_f\right) \quad (6)$$

where the superscript n represents the time level, the subscripts i,j and Δx, Δy are the cell position and the grid size in x and y directions, respectively, and Δt^n is the timestep calculated accordingly to the Courant–Friedrichs–Lewy condition. The fluxes $\mathbf{F}$, $\mathbf{G}$ in x and y-directions are evaluated at the intercells $i \pm 1/2$ and $j \pm 1/2$, respectively.

To avoid the formation of high velocities and instabilities, if the water depth $h_{i,j}$ is lower than a small threshold h_ε the cell is dried ($h = 0$).

The second order of accuracy in time is obtained by a second order Runge–Kutta method:

$$\mathbf{U}_{i,j}^{n+1} = \mathbf{U}_{i,j}^{n} + 0.5\Delta t^n[\mathbf{D}_i(\mathbf{U}_{i,j}^{n}) + \mathbf{D}_i(\mathbf{U}_{i,j}^{n+1/2})] \quad (7)$$

where the operator $\mathbf{D}_i(\mathbf{U}_{i,j})$ is defined as:

$$\mathbf{D}_i(\mathbf{U}_{i,j}) = -\frac{\left(\mathbf{F}_{i+\frac{1}{2},j} - \mathbf{F}_{i-\frac{1}{2},j}\right)}{\Delta x} - \frac{\left(\mathbf{G}_{i,j+\frac{1}{2}} - \mathbf{G}_{i,j-\frac{1}{2}}\right)}{\Delta y} + \mathbf{S}_0 + \mathbf{S}_f \quad (8)$$

and $\mathbf{U}_{i,j}^{n+1/2}$ is obtained as:

$$\mathbf{U}_{i,j}^{n+1/2} = \mathbf{U}_{i,j}^{n} + \Delta t^n\mathbf{D}_i(\mathbf{U}_{i,j}^{n}) \quad (9)$$

3. GPU modeling of Shallow Water Equations

3.1. GPU architecture

We start this section with a brief overview of the main features of a GPU's architecture (see Kirk and Hwu, 2010 for further reading and cud (Cuda best practices guide); Brodtkorb et al., 2013 for best programming practices). The *Compute Unified Device Architecture (CUDA)*, adopted by our system, is a framework for GPU-based parallel computing introduced by NVIDIA™.

CUDA is a high-level language that can be partially run on a GPU. The CPU (the *host*) and the Graphics Process Unit (the *device*) can be controlled by a CUDA program that can take advantage of both resources.

In order to create a uniform and independent view of the hardware, a logical representation of the computation is used. The basic work unit is the *thread* and many threads are grouped into *blocks*. Since the hardware architecture contains a variable set of multiprocessors and each of them can run a number of parallel threads, the blocks can be actually processed by different processors (real parallelism) or queued by the GPU's scheduler, if the number of blocks is larger than the processors available. Moreover, each block is typically processed by groups of 32 concurrent threads (named a *warp*), which represent the smallest unit of parallel work on the device. In conclusion, large tasks are typically fragmented over thousands of logical blocks and threads, while the GPU scheduler provides a lightweight switch among them.

The computational model used by GPUs is SIMD (Single Instruction Multiple Data); therefore, the same set of instructions is executed by every thread in a warp. If this does not happen (e.g., two threads execute different branches of an `if-then-else` construct) the warp is fragmented into two different execution paths and each of them is assigned to a new warp, with a consequent sub-utilization of the parallel processors.

A *kernel* is the piece of code that is executed by the GPU. When the CPU invokes a kernel, it specifies the number of blocks and threads defining the GPU organization of tasks and the kernel's arguments are passed to GPU.

Due to the original purpose of the GPU, which is a graphical pipeline for image rendering, the GPU memory architecture is rather different from the standard RAM–cache–registers hierarchy. It features six different levels of memory, with different properties in terms of location on chip, caching, read/write access, scope and lifetime: *(1)* registers, *(2)* local memory, *(3)* shared memory, *(4)* global memory, *(5)* constant memory, and *(6)* texture memory. In particular, registers and local memory have a thread life span and visibility, while shared memory has a block scope (to facilitate thread cooperation); the other memory levels are permanent and visible from host and every thread on the device. Constant and texture memories are the only memories to be read-only and to be cached.

Usually, the design of data structures for efficient memory access is the key to achieve good speedups, since access time and bandwidth of transfers are strongly affected by the type of memory and by the sequence of access patterns by threads (coalescing). Only registers and shared memory provide low latency, if shared accesses are either broadcasts (all threads read same location) or conflict-free accesses. The shared memory is divided into 16 different memory banks, which can be accessed in parallel. When two different threads request the same bank, then the requests are serialized, thus reducing the degree of concurrency.

3.2. CUDA implementation of SWE

The modeling a parallel system based on a GPU architecture requires some adaptation of traditional sequential algorithms. In this section we refer to second order simulations, even if the system can also work parametrically on first order ones.

The general structure of the computation is summarized in Fig. 1, where the update of the conservative variables (vector **U**) in a single timestep from time t^n to time $t^{n+1} = t^n + \Delta t$, (as defined in Equation (6) for first order models and in Equation (7) for second order ones) is described by a sequence of eight sub tasks. The CPU controls the iteration evolution and invokes the various tasks in order. Each task can be processed in parallel and it is described by a distinct CUDA *kernel*. Tasks cannot be further parallelized and/or merged because of data dependencies. At the end of each timestep, a communication phase between CPU and GPU is performed and particular care is devoted to reduce the bandwidth of such exchanges. The conservative variables are transferred from the device to the host, only for periodical file writing and/or debugging. The system is also equipped with a OpenGL visualization module, which periodically displays the evolution and additional debugging information. When the visualizer is activated, data already on the GPU's memory are migrated to the rendering pipeline without CPU intervention.

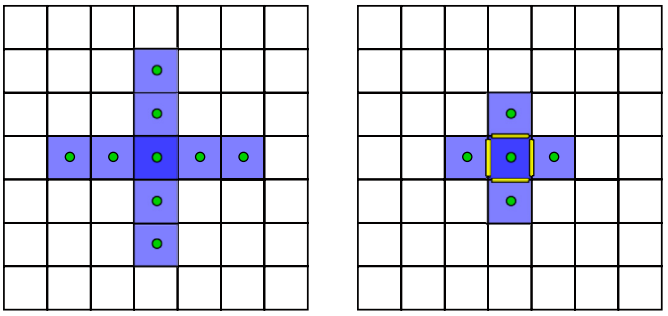

Fig. 2. Access pattern with (left) global ghost cells and (right) implicit local ghost cells. The yellow rectangles represent the left and right reconstructions at intercells. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The resulting execution pattern is an interleaved usage of CPU and GPU. The second order simulation requires two half steps computations (sub-tasks 3, 4 and 5, 6 respectively). In case of a first order simulation, sub-tasks 3, 5 and 7 are skipped and the update in time is performed in the sub-task 4.

Compared to Brodtkorb et al. (2012) where the MUSCL reconstruction, flux computation and integration are performed in a single kernel, our design allows to split the computation into two separate kernels. In particular, the first one performs the MUSCL reconstruction and the second one computes the fluxes and updates the conserved variables. As shown in Fig. 2 this affects also the underlying data structures to store temporary data (i.e., fluxes) and the accesses to them: in the approach used by Brodtkorb et al. (2012), the kernel gathers the conserved variables of 8 neighbor cells in order to compute both the left and right MUSCL reconstruction on each edge. Following the approach herein proposed, on the contrary, both in the MUSCL reconstruction kernel and the integration one, only the values of the 4 neighbors cells have to be read by each thread of the kernel, therefore both kernels require more compact stencils for the computation. As explained in details in Section 3.4, this has a remarkable impact on the possibility to adopt more flexible boundary conditions without increasing the code branching.

In the remainder of the section we present the main data structure organization and a description of the above-mentioned tasks.

3.3. Logical organization

The data organization relies on a simple, yet efficient, dynamic schema. Since we adopt a Cartesian mesh, a straightforward

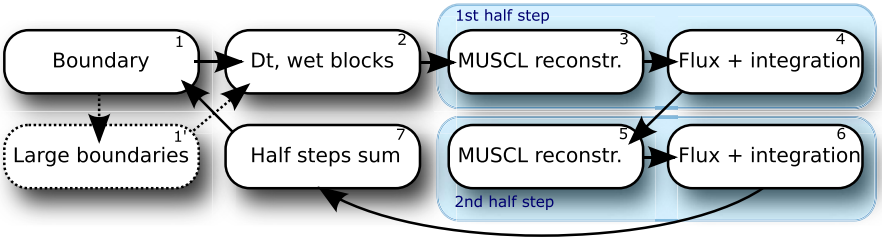

Fig. 1. Iteration flux diagram. Each phase is handled by a GPU kernel.

organization of logical blocks defines a correspondence between cells and threads. In particular, a block is defined over the two dimensional space and, in turn, covers a squared tile of the computational mesh. Each cell in the tile (identified by a two-dimensional index) belongs to a block and it is processed by a unique thread. Each block is identified by its position in the mesh and can take advantage of the locality of data. In particular, coalesced retrieval of information (contiguous threads read contiguous areas of memory) and cached data (textured matrices) provide efficient parallel speedups when the same code is run among the whole block. In our application the data matrix is partitioned into two-dimensional blocks of size 16×16, which empirically provided the best performances. The whole processing is based on such blocks.

To improve the efficiency, we dynamically select which blocks should be activated and processed at each timestep, extending the idea presented by Brodtkorb et al. (2012): the set of *active blocks* $\mathscr{A}$ are defined as the set of blocks that contains at least one wet cell and/or one cell that may become wet at the end of the timestep and thus are the blocks that are involved in potential changes at that time. Active blocks at time t^n are the only ones processed by GPU kernels. The Block Deactivation Optimization (BDO), which accurately estimates the minimum set of blocks that may become wet within the next timestep is presented in Section 3.6.

Boundary conditions may exist inside an active block: in this case a specific code differentiates the execution with respect to other cells. Whenever a kernel execution contains different flows of code, the GPU scheduler separates the threads in different runs, with a decrease of parallel use of the GPU's cores. However, the number of cells involved in boundary conditions is fixed and is proportional to the perimeter of the basin; for practical test cases, it is usually much smaller than the total number of cells. In the next section we provide more details about the organization of boundary information.

3.4. Implicit local ghost cells

The implicit local ghost approach used by our implementation corrects the neighbors' values of a cell, according to the presence of a boundary condition. Basically, every time a cell is read along with related boundary information, the code is instructed to simulate the correct values for neighbors. This differs from the global ghost cell approach (Brodtkorb et al., 2012), where ghosts cells are explicitly modeled in the grid. By contrast in our approach, ghost values are computed at each iteration and they are not stored on the grid. The advantage of this implementation is that complex boundary geometries can be represented without any conflicts (e.g. a wall boundary condition on the north edge and a discharge boundary conditions on the west edge for the same ghost cell).

It has to be underlined that this approach is possible only because the code was designed in such a way that each kernel requires information of one neighbor per side (Fig. 2-right), and this, opposite to the approach adopted by Brodtkorb et al. (2012) (Fig. 2-left), dramatically reduces the code branching if complex boundaries are present. The information about potential boundaries around each cell into an additional matrix is stored in GPU's memory. In particular, for each grid cell, information about the existence and type of boundary condition for north, south, east and west neighbors is stored in a textured memory (see, e.g., Fig. 3 on the right, the red line). This information is accessed during kernel execution and it activates specific instructions that impose the implicit local ghost equations. This requires that each kernel (both MUSCL reconstruction and integration ones) is extended with a piece of code that handles every type of boundary condition associated to each side of the cell. This approach differs from the global ghost cell one, where a specific kernel at the beginning of each timestep updates the conserved variables in the ghost cells. The extra burden of our design is paid off by the flexibility of complex boundary conditions and the possibility to model real world simulations. Compared to global cell approach, our implementation requires access to cell's boundaries information record (one textured read from boundary matrix) and to one neighboring cell per side.

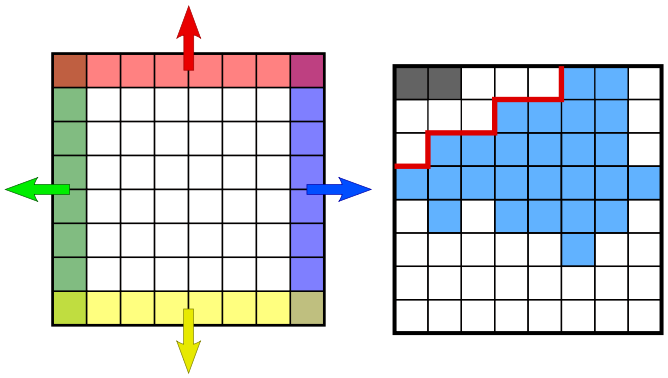

Fig. 3. (left) Block Deactivation Optimization. If a colored cell is wet, the adjacent block is activated at next iteration, since it may contain a new wet cell. (right) Example of wet cells (blue), boundary cells (green), boundary neighbors (red line) and outlier cells (gray) are given. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Another advantage of the algorithm herein adopted is that wet and dry cells belonging to an active block are processed by the same set of instructions: in this way the number of the code branches is minimized exploiting the parallel capability of the GPU.

The SWE simulation requires a number of matrices to be stored in GPU's memory. Given the potentially large size of the matrices, global memory is chosen. The basic vector of data is assumed to be a structure of 4 floats, named *gridpoint* that stores the quadruple (h, uh, vh, z).

3.5. Open boundary conditions

If open boundary conditions are activated, then some conserved quantities have to be imposed at the boundary cells, according to the characteristics of the flow. In particular, for inflow boundary conditions, the specific discharge is always imposed, whereas the water depth is added only if the flow is supercritical. Differently, for outflow boundary conditions, no quantities are assigned if the flow is supercritical, whereas the water elevation (or a stage-discharge relationship) is imposed when the flow is subcritical.

When quantities have to be assigned, the system input is the specific discharge and/or water elevation at discrete time samples. At the current simulated time, these quantities are linearly interpolated by the CPU and passed to GPU. For each open boundary condition, the boundary kernel is launched over the blocks that are covered by the condition. For each block, the kernel updates the conserved variables $\mathbf{U}$ in the cells involved in the condition. Since each thread is mapped to a cell and few cells in a block require updates, most of threads will immediately terminate. For an inflow boundary condition the specific discharge is redistributed along the cells of the string involved as follows:

$$(uh)_k = \frac{(uh)_{bc} N}{\sum_{k=1}^{N} h_k^{\chi}} h_k^{\chi} \tag{10}$$

where $(uh)_{bc}$ is the initial specific discharge that needs to be imposed, N is the total number of cells of the string, the exponent χ

is assumed equal to 5/3 in the present work according to the Chezy–Manning equation and $(uh)_k$ is the specific discharge obtained for the k-th cell involved in the inflow boundary condition. This re-normalization is applied to avoid unphysical velocities in cells with small depths and to ensure a more realistic velocity distribution at the inflow. However this procedure is endowed with one operational caveat: since inflow boundary conditions require a re-normalization among all cells involved, it is necessary to first compute the summation on the right-hand side of Equation (10) over all cells. Unfortunately, if the cells span among different blocks, the summation cannot be performed by a single kernel. Due to the hardware architecture, different blocks cannot share information during a kernel execution, therefore this simple operation is implemented by two kernels running in sequence. The first one (in boundary) computes partial summations, one for each block; then, at kernel completion, the CPU retrieves from GPU's memory values computed by each kernel and combines them into the final value. In case of boundaries spanning over more than one kernel, the CPU invokes the second kernel, which divides all cells in all blocks by the value passed by the CPU. For cells covered by a single block, the normalization can be performed within the boundary kernel.

The technique used to collect the summation is based on an efficient version of logarithmic reduction, which is able to process 2^n elements in n runs of 2^n, 2^{n-1}, 2^{n-2}, … threads. This technique is used every time a block needs to communicate some values to CPU; indeed it avoids passing to the CPU, processes sequentially a set of 2^n elements and finally it can use parallel work to speedup the process down to n parallel steps. Moreover, the bandwidth for transferring data is highly reduced, while a negligible CPU overhead to combine results of each block is still required.

3.6. Timesteps Δt and Block Deactivation Optimization

This kernel performs two independent tasks. In order to improve the efficiency, the kernel is designed to operate incrementally: the active blocks $\mathscr{A}$ computed at previous timestep are the basis for the next operations. It is worth recalling that a block is active when it contains at least one wet cell and/or one cell that might become wet at the end of each timestep. At the beginning of the simulation (first timestep) the kernel processes the whole domain or, in other words, $\mathscr{A}$ coincides with the complete set of blocks.

The goals of the kernel are: (a) computing the Δt timestep according to the Courant–Friedrichs–Lewy condition and (b) computing the set of active blocks $\mathscr{A}'$ which are wet and/or may become wet during the timestep. For each block, the kernel simply computes the Δt of each cell in the block, using the standard mapping of one thread to one cell. The block computes the minimum value found, by a logarithmic reduction (see previous section) and stores it in a buffer in global memory. At the end of the kernel execution, the CPU collects from GPU's memory the Δt returned by each block and retrieves the minimum.

The computation of the new set of active blocks is based on the current set $\mathscr{A}$. Being the scheme explicit, the simple observation that after one timestep the wet/dry front can evolve by at most one cell, provides a property that can be used to estimate whether it is necessary to add a neighbor of an already active block (see Fig. 3 on the left). The kernel processes the active blocks and checks the presence of at least one wet cell along each of the four inner borders of the blocks (see colored cells in the figure). Any partially wet border may generate a wet cell in the confining block during the next timestep. In Practice, the kernel is able to perform four reductions over each border and to return this information to CPU. In addition, the computation of the Δt allows to establish when the block is completely dry. When the kernel is completed, this information is processed by the CPU and the new set $\mathscr{A}'$ is computed according to the data passed by single blocks. When compared to Brodtkorb et al. (2012) this strategy allows to reduce the number of unnecessary active blocks, thus enhancing performances. Fig. 3 (right) shows an hypothetical configuration where wet cells cover part of a block. Using the Block Deactivation Optimization, it results that the southern neighboring block should not be activated, since no wet cells are present in the lower row of the block (cfr. the yellow area in the left figure).

3.7. Monotone Upstream-centered Scheme for Conservation Laws (MUSCL) reconstruction

This kernel is required by second order accurate numerical scheme and computes, for all cells in the active blocks, the reconstructed values of the conserved variables at the west/east/north/south edges according to the Monotone Upstream-centered Scheme for Conservation Laws (MUSCL) depth-positive procedure (see Appendix A). Some blocks can contain boundary or even cells outside the computational domain, therefore the texture with this information is read. In case of boundary, specific branches are applied to correct the values and the reconstructions are then written in GPU's global memory. This kernel cannot perform the integration step directly, since in case of cells located at block borders, reconstructions for neighboring blocks are needed. Consequently, the kernel needs to complete the computation for each block and, after its completion, the CPU invokes the next integration kernel.

3.8. Flux computation and time integration

The Flux computation and time integration kernel exploits the cached access to GPU's memory thorough textures when accessing to the reconstructed values, which have been computed by the previous kernel. In case of first order simulation no spatial reconstruction is operated (and the MUSCL reconstruction kernel is skipped): values of the conserved variables at the cell edges are identical to the conserved variables vector $\mathbf{U}_{i,j}$ at the center. The reconstructed values are then used to compute the four numerical fluxes for the cell (i, j): $\mathbf{F}_{i+½,\,j}$, $\mathbf{F}_{i-½,\,j}$, $\mathbf{G}_{i,\,j+½}$ and $\mathbf{G}_{i,\,j-½}$ according to the HLLC approximate Riemann solver (Aureli et al., 2008) in both first and second order schemes. It has to be noted that in the present algorithm the same flux at each edge is actually computed twice by two different threads, since, e.g. the west flux for cell (i, j) is equal to the east flux for cell $(i-1, j)$. An alternative solution could have been to compute fluxes by an additional kernel (which associate one thread to the edge of the grid), store the fluxes values on global memory and read them by the next kernel which should have just updated the conserved variables. On the GPU architecture, however, the adopted approach is more efficient because the penalty of accessing global memory and extra memory consumption is largely outperformed by the small computational overhead. Moreover, the other approach would require an additional kernel for the explicit computation and writing of the reconstructed values.

After fluxes have been computed, the vector of conserved variables $\mathbf{U}_{i,j}$ is updated by the same kernel. In the first order numerical scheme the Equation (6) is adopted. In the second order numerical scheme the conserved variables are updated by Equations (9) and (8), in the first (task 4) and second half timesteps (task 6). If the second order accurate scheme is used, both the $\mathbf{U}_{i,j}^{n+1/2}$ and the $\mathbf{D}(\mathbf{U}_{i,j})$ have to be stored in the global memory at the end of the first half timestep, whereas only $\mathbf{D}(\mathbf{U}_{i,j}^{n+1/2})$ is stored for the second half timestep. To improve efficiency, a double buffering technique is employed to store the new conserved variables matrix, while keeping the previous values for textured reading purposes.

4. Test cases

4.1. GPU details

In this section, we present and discuss computational experiments on several test cases, based on different graphic cards. In particular a low-end (GTS 450), a mid-end (GTX 580) and a high-end (Tesla M2070) video cards were used. Table 1 reports basic specifications (best values are highlighted in bold font) and a rough price estimation. We would like to point out some relevant aspects of the cards in use. The GTS 450 card represents an example of a simple video card that can be easily found in a standard workstation, this card has been tested to show the impact of parallel simulation with almost no investments. The GTX 580 card is dedicated to top performances in videogames and therefore features a high number of cores with high clock speed. The Tesla M2070 card, on the other hand, delivers a large RAM (4 times more than the GTX 580) to accommodate larger scientific data for processing. However, its number of processors and especially its clock rate are lower than the GTX 580. The higher price of this card is mainly justified by the memory mounted on the device and by the architectural design.

A closer analysis of the hardware can lead to the choice of the most suitable device for computation. In the test cases presented, we run the simulations with the three cards and compare performances. As a guideline, it is expected that the only candidate for large simulations with high memory requirements is the Tesla M2070 (or equivalent), while the top performances in terms of speed can be reached by the GTX 580 card.

4.2. Performance metrics

The performances of the test cases will be measured by means of the index named Megacells/s (or, in short, MC/s). This index captures the number of cells (in millions) divided by the computational time required to process them. Please note that, in order to obtain a more robust measure, both the number of cells and the computational times are calculated since the beginning of the simulation.

Higher MC/s values correspond to higher performances, since more cells are computed in the same time reference window. In our measurements a cell in the grid is considered as processed, regardless its dry/wet status. Therefore, an effective Block Deactivation Optimization affects the measure by an increase of the MC/s values, since the actual number of operations to compute an iteration decreases. This measure allows to capture the average speed during the computation and it is particularly useful in case of real domains, where the number of wet cells can significantly be modified along the simulation time. We believe that this measure provides more insights than a speedup ratio (sequential computational time over parallel time). In fact, the compared source codes (sequential against parallel) are inherently different and, additionally, the ratio suffers from a bias depending on the sequential CPU model being adopted. Therefore, we report a MC/s measure, which also provides a quantitative measure about the speed of the computation. Nevertheless, for some test cases the speedup is also reported with respect to a specific sequential processor and sequential implementation.

4.3. Wet cylindrical dam break

To assess the performance of the code, we run the classical cylindrical dam break test case (Toro, 1999a,b) with a flat frictionless bottom. A cylindrical water volume of radius $R_0 = 10$ m is initially placed in a circular domain with radius equal to 25 m centered in ($x_0 = 0$ m, $y_0 = 0$ m) As initial condition: $\eta(r, 0) = 10$ m if $0 \le r \le R_0$ and $\eta(r, 0) = 1$ m elsewhere, where $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$. At the beginning of the simulation the fluid is at rest. Since the problem has a cylindrical symmetry, an inhomogeneous set of 1D differential equations can be derived along r (Toro, 1999a,b). This set of equations is solved on a very fine mesh with cell size $\Delta r = 0.005$ m in order to obtain the reference solution. Conversely, in the 2D simulation the computational domain was discretized through a square grid with cell size $\Delta x = 0.1, 0.05, 0.025$ m, as shown in Table 2. All simulations have been performed until time $t = 1$ s.

Fig. 4 shows the water surface elevation at time 0.4 s obtained with the first and second order accurate numerical scheme and with a grid size $\Delta x = 0.05$ m. The results are in agreement with the reference solution; both first and second order models are able to reproduce the shock position. As expected, the rarefaction wave obtained by the second order model is closer to the reference solution. Table 2 shows the Megacells/s obtained using three different GPUs. Since the domain is completely wet, the MC/s values are constant throughout the computation for a specific GPU. The results show an almost constant Megacells/s ratio for a given GPU among different grid sizes (within 5% variability) and this also means that notwithstanding the increased simulation size, the throughput (MC/s) is unaltered and not affected by potential parallel bottlenecks. Therefore a good design of the code and scalability can be assessed by these tests for all the GPUs. Different hardwares with different number of cores have a strong impact on performances. Indeed it can be noted that for larger cell sizes performances slightly decrease, mainly because of the larger ratio of boundary cells versus inner cells.

In fact, the treatment of boundary cells is less efficient than the treatment of inner cells, because it requires more asymmetric work and therefore causes a light penalty. So, if the ratio $r_b =$ (number of boundary cell)/(total number of cell) is higher, then the code is less efficient. It is expected that the number of inner cells is quadratic with respect to the number of boundary cells. Therefore the ratio r_b decreases with the cell size thus reducing the computational impact of boundary treatments. If we compare two different simulations (1 and 2) of the same test case with grid sizes Δx_1 and $\Delta x_2 = 0.5\Delta x_1$, then it can be easily shown that $r_{b2} = 0.5 r_{b1}$. From the computational point of view, in simulation 2 there will be proportionally less boundary condition cells, and therefore their handling will have a lighter impact on a single iteration. This means that the simulation with the higher resolution (simulation 2) is

Table 1
GPU card specifications.

Card	GTS 450	GTX 580	Tesla M2070
Cores	192	**512**	448
Processor clock (MHz)	783	**1544**	575
GPU RAM (GB)	1	1.5	**6**
Bandwidth (GB/s)	57.7	**192.4**	150
Est. price ($, 2013)	**100**	400	1000

Table 2
Wet cylindrical dam break: Megacells/s for First Order (FO) and Second Order (SO) schemes at time $t = 1$ s.

	GTS 450		GTX 580		Tesla M2070	
Δx	FO	SO	FO	SO	FO	SO
0.1 m	45.75	17.61	173.29	66.58	145.41	53.12
0.05 m	47.75	19.08	187.98	71.27	161.22	57.78
0.025 m	48.49	19.18	194.92	64.30	166.94	57.21

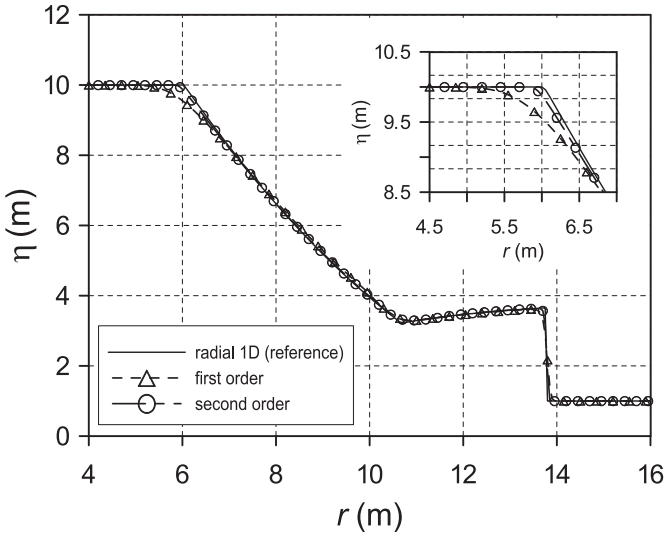

Fig. 4. Cylindrical dam break with wet bed: water surface elevation with first and second order FV model at time 0.4 s with $\Delta x = 0.05$ m, compared against the reference 1D radial solution with $\Delta r = 0.005$ m.

more efficient (higher value of Megacells/s, see Table 2) than the one with lower resolution. Please note that this does not mean that the computational time of simulation 2 is less than the one of simulation 1. If we define t_{cpu} the computational time required to update a timestep of the solution and we assume that the efficiency (Megacells/s) is constant in both simulations, then $t_{cpu,2} = (\Delta x_1/\Delta x_2)^2 t_{cpu,1}$, since we need to account for the actual number of cells in the simulations.

The second order simulation performs roughly 3 times slower than the first order one. This is justified by the fact that in the second order simulation three extra kernels (two MUSCL reconstructions and the integration at half timestep) must be invoked and they require similar parallel time. The CPU runtime for grid size $\Delta x = 0.05$ m is equal to 0.4 MC/s for the serial version of the code (runtime 1749 s for 1 s of physical time) with an Intel Xeon CPU X5365 3.00 GHz and using the second order accurate model. This setup represents a typical workstation capability in 2010. The speedup for this particular test case varies from 49 (for the GTS 450 GPU) to 178 (for the GTX 580 GPU).

As already highlighted in Section 4.2, the speedup gives only a rough approximation of the code efficiency since the estimate is biased by the CPUs architecture in use. In the remainder of the section we mainly focus on MC/s values for different GPUs.

4.4. Cylindrical dam break over a dry non-flat bottom

In order to test the numerical model in the presence of non-flat bottom and of wet/dry fronts, a cylindrical dam break with non-flat bottom has been simulated (Aureli et al., 2008). A cylindrical water volume of radius $R_0 = 10$ m is initially placed in a circular domain of radius $R = 25$ m centered in ($x_0 = 0$ m, $y_0 = 0$ m). The bottom profile is described by the following equation:

$$z(r) = 0.5\left[1 + \cos\left(\frac{2\pi}{5}r\right)\right], \tag{11}$$

where $r = \sqrt{(x-x_0)^2 + (y-y_0)^2}$ is the radius.

As initial conditions it is assumed:

$$\begin{aligned} \eta(r,0) &= 10\ \text{m} \quad \text{if } 0 \le r \le R_0 \\ h(r,0) &= 0\ \text{m} \quad \text{if } R_0 < r \le R \\ u_r(r,0) &= 0\ \text{m/s} \quad \text{everywhere.} \end{aligned} \tag{12}$$

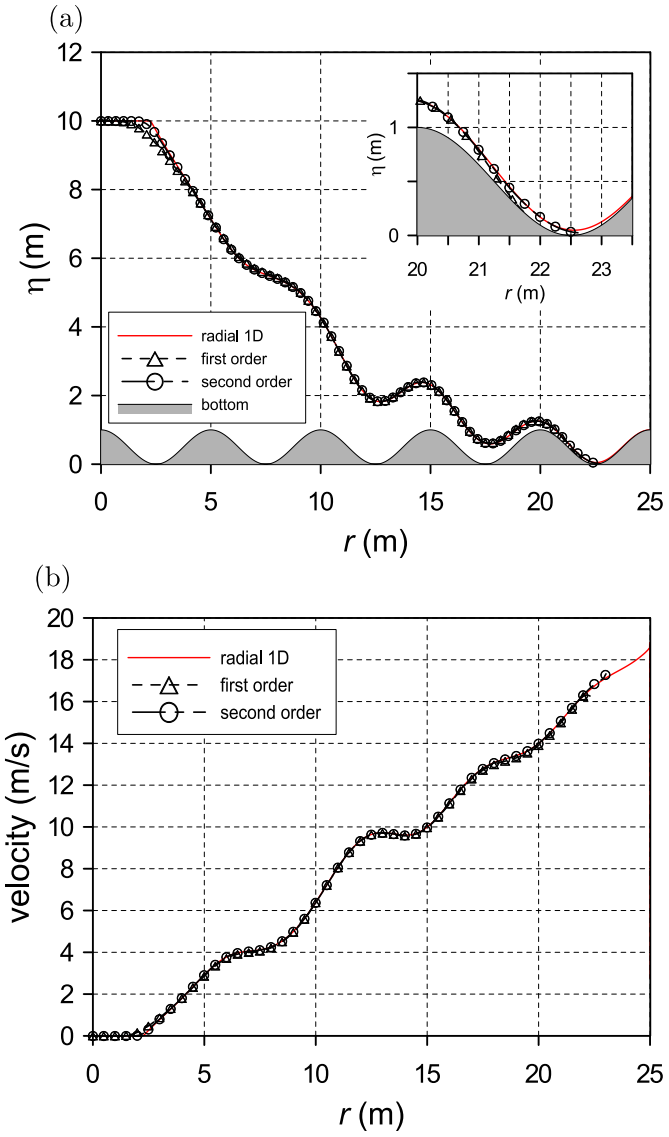

Fig. 5. Cylindrical dam break with non-flat dry bed: (a) water surface elevation and (b) velocity with first and second order FV model at time 0.8 s with $\Delta x = 0.05$ m compared against the radial 1D reference solution with $\Delta r = 0.005$ m.

As discussed in the previous section, the reference solution is obtained by solving the radial 1D set of equations on a very fine mesh with size $\Delta r = 0.005$ m. The simulation is run with grid sizes $\Delta x = 0.1$, 0.05 and 0.025 m and with a wet-dry threshold value $h_\varepsilon = 5 \times 10^{-4}$ m for 1 s of physical time.

Fig. 5 shows the water surface elevation and the velocity at section $y = 25$ m and at time 0.8 s, obtained with $\Delta x = 0.05$ m. Both first and second order solutions are in agreement with the reference solution; as expected the second order guarantees that the wet-dry interface and the rarefaction wave are closer to the reference solution. The depth-positive reconstruction (Appendix A) ensures an accurate description of the wet-dry moving front with no spurious oscillation. The capability of the schemes (both first and second order) to maintain the C-property can be appreciated close to the center of the domain. Fig. 6 shows the Megacells/s obtained by using three different GPUs, and Fig. 7 shows the ratio between the number of dry cells observed during the simulation and the total number of cells used to discretize the domain: at the beginning of the simulation most of the domain is dry, whereas at

time 0.89 s the domain is completely wet. In this case the Block Deactivation Optimization procedure (see Section 3.6) is activated, and it is able to constrain the computational efforts over a limited area. This can be noticed by the rather high MC/s values obtained in the first part of the simulation (see Fig. 6), which decrease with a trend correlated to the number of wet blocks in the simulation. The MC/s stabilizes when the simulation reaches a completely wet surface. It can also be noticed that the performances are slightly

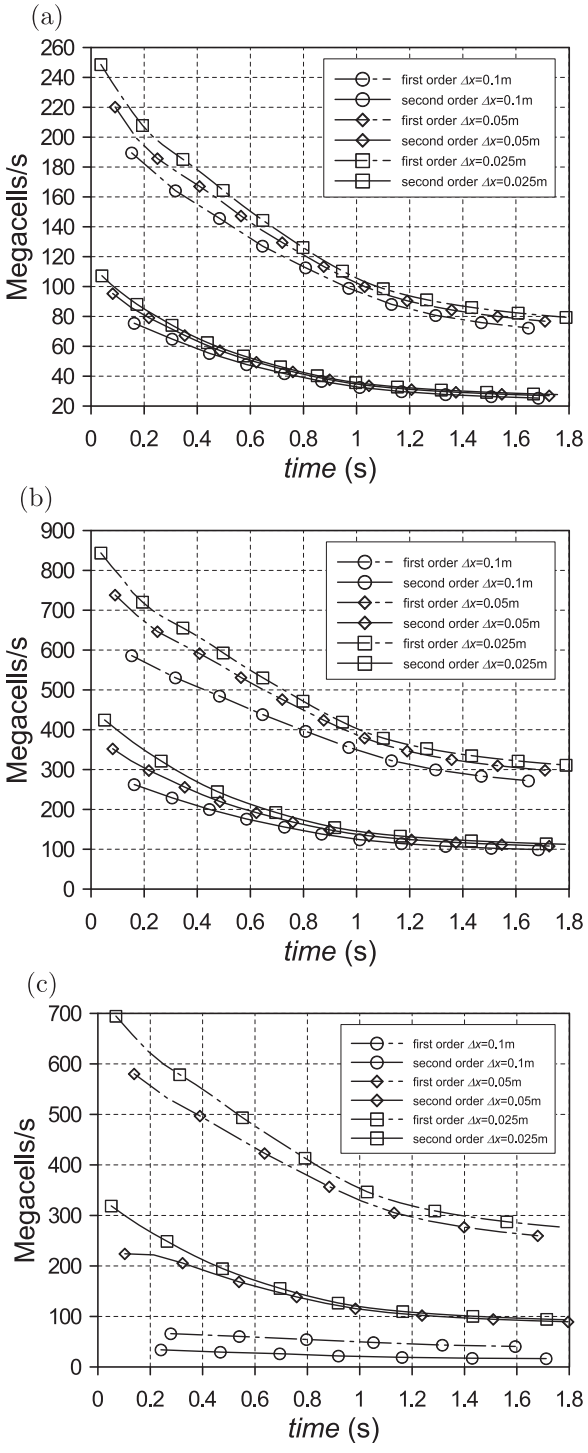

Fig. 6. Cylindrical dam break with non-flat dry bed: Megacells/s obtained with (a) GTS 450, (b) GTX 580 and (c) Tesla M2070.

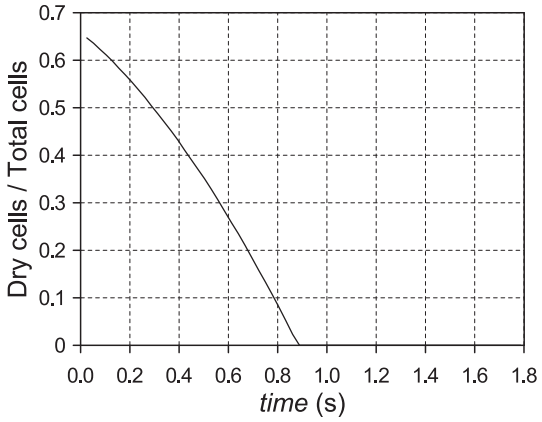

Fig. 7. Cylindrical dam break with non-flat dry bed: ratio between number of dry and total cells observed during the simulation.

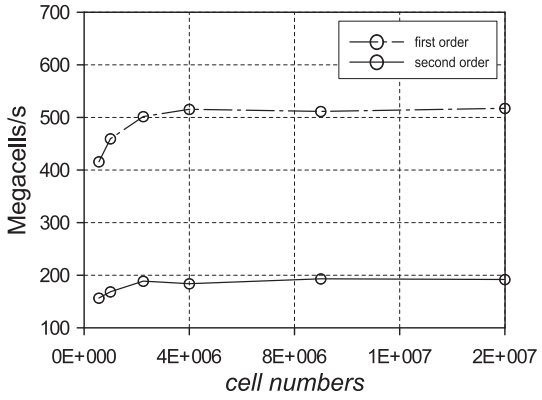

Fig. 8. Cylindrical dam break with non-flat dry bed: Megacells/s at time 0.5 s with a Tesla M2070.

influenced by the cell size. For the Tesla M2070 GPU, the differences are more noticeable and we investigate this in depth. In Fig. 8 the Megacells/s at time 0.5 s for the Tesla M2070 GPU are shown for simulations with different grid sizes. In the figure, the total number of cells for each simulation is reported.

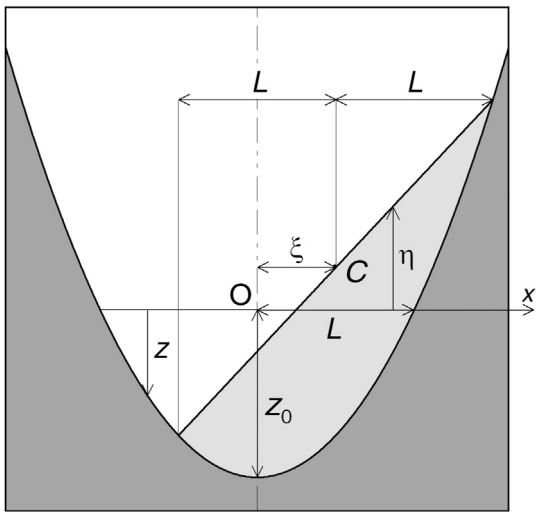

Fig. 9. Definition sketch for Thacker test with planar water surface (Thacker, 1981).

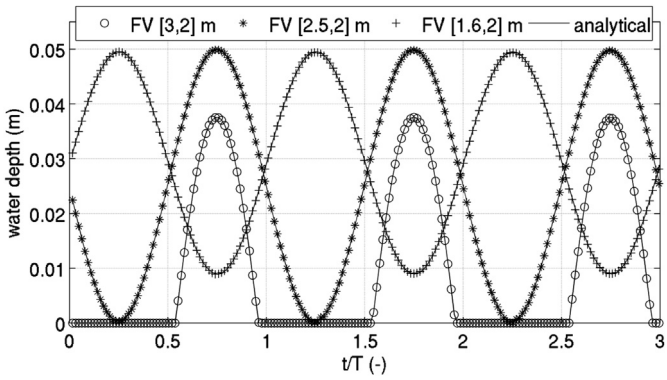

Fig. 10. 2D oscillating planar water surface: water depth at points with coordinates [x, y] = [3.00, 2.00] m, [2.50, 2.00] m and [1.60, 2.00] m.

Table 3
2D oscillating planar water surface: Megacells/s for First Order (FO) and Second Order (SO) schemes.

	GTX 580		Tesla M2070	
Δx	FO	SO	FO	SO
0.01 m	428	169	324	139
0.005 m	562	211	441	177
0.0025 m	631	227	573	207

Figs 6 and 8 show that the Megacells/s increase with the resolution: simulations with more cells have a higher Megacells/s. As pointed out in Section 4.3, this is due to the ratio between the number of boundary cells and the total number of cells, which is not constant with the resolution. Moreover, with coarser resolutions, there is a smaller number of blocks processed by the GPU and

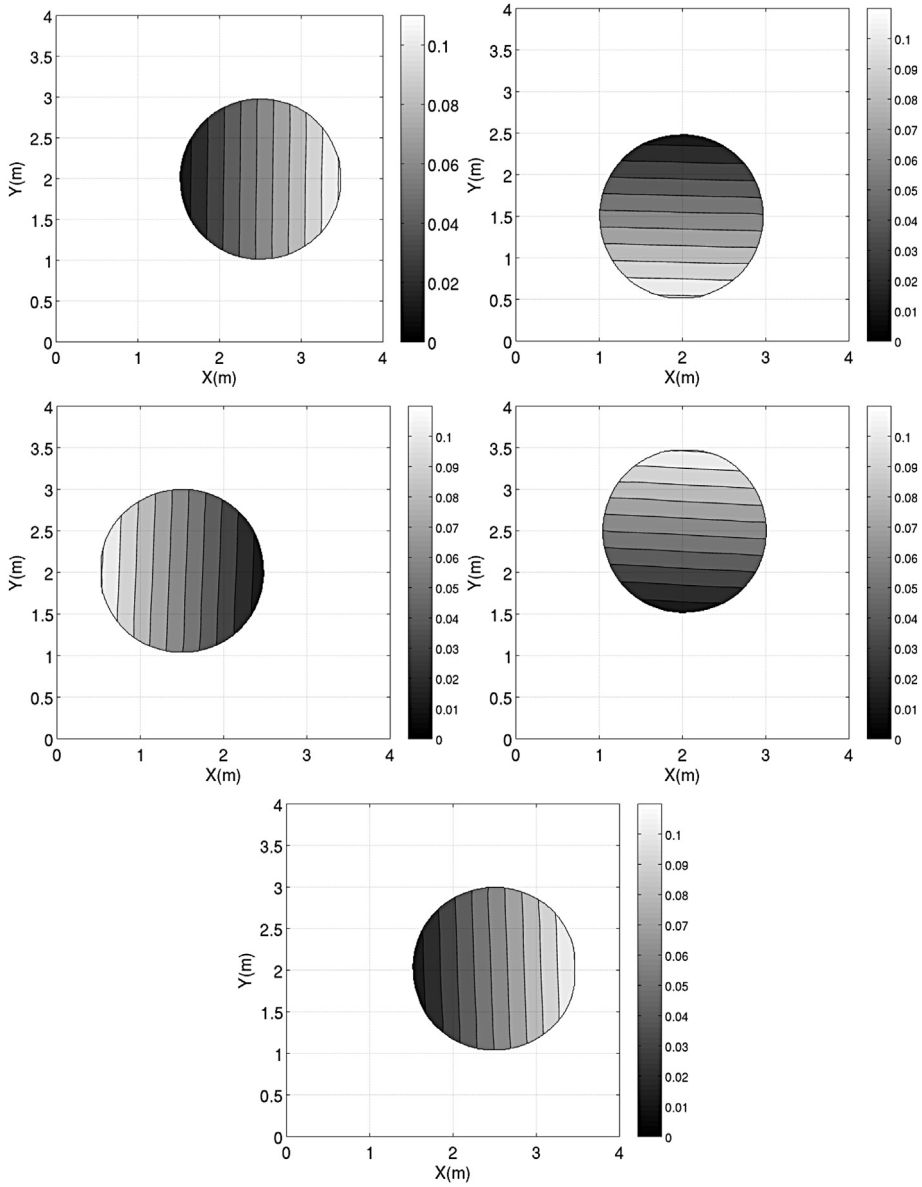

Fig. 11. 2D oscillating planar water surface, water free surface elevation maps at times t/T 2.00, 2.25, 2.51, 2.76 and 2.99.

this does not compensate the memory exchange overhead and cannot keep the cores busy for enough time. The parallel speedup, however, is not affected by larger number of cells and proved to be very robust for heavier tasks. The current limitation is provided only by the maximal GPU memory that can accommodate the problem. The CPU runtime, considering $\Delta x = 0.05$ m obtained with the serial version of the code, is equal to 1.3 Megacells/s (runtime 277 s for 1 s of physical time) with an Intel Xeon CPU X5365 3.00 GHz and using the second order accurate model. The speedup for this particular test case varies from 18 (for the GTS 450 GPU) to 74 (for the GTX 580 GPU).

4.5. 2D oscillating planar water surface

The capability to provide accurate results in the presence of 2D wetting and drying moving boundaries is essential in order to simulate real test case problems. For the proposed method, we compare numerical results with the exact solution given by Thacker (1981), which predicts the oscillation of a water volume in a frictionless paraboloid basin according to the equation:

$$z = z_0\left(1 - \frac{x^2 + y^2}{L^2}\right). \tag{13}$$

As shown in Fig. 9, in Equation (13) the depth function z is positive below the equilibrium level, z_0 is the depth of the vertex of the paraboloid and L is the radius at $z = 0$. In the particular case considered, the water body is initially planar and the velocity field is uniform; the analytical solution and the initial conditions are given by:

$$\begin{cases} \eta(x,y,t) = 2\xi\frac{z_0}{L}\left[\frac{x}{L}\cos\omega t - \frac{y}{L}\sin\omega t - \frac{\xi}{2L}\right] \\ u(x,y,t) = -\xi\omega\sin\omega t;\ v(x,y,t) = -\xi\omega\cos\omega t, \end{cases} \tag{14}$$

where η is the surface elevation (positive above the equilibrium level) and $\omega = \sqrt{2gz_0}/L$ is the frequency of the rotation around the center of the basin. The magnitude of the velocity is constant over time at the value $|\mathbf{v}| = \xi\omega$, whereas the direction rotates over time.

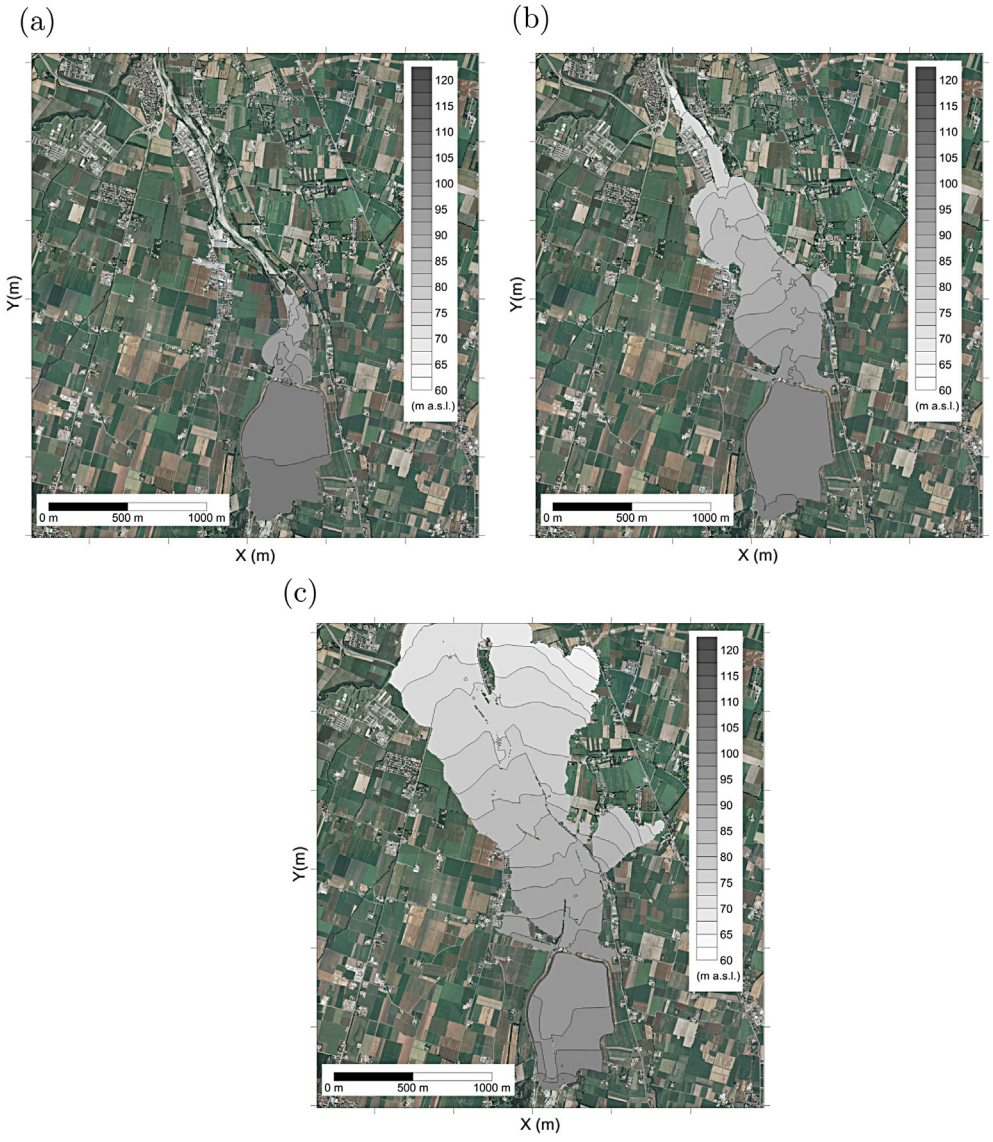

Fig. 12. Dam-break of Parma flood control reservoir, water free surface elevation maps at times (a) 2, (b) 10 and (c) 30 min.

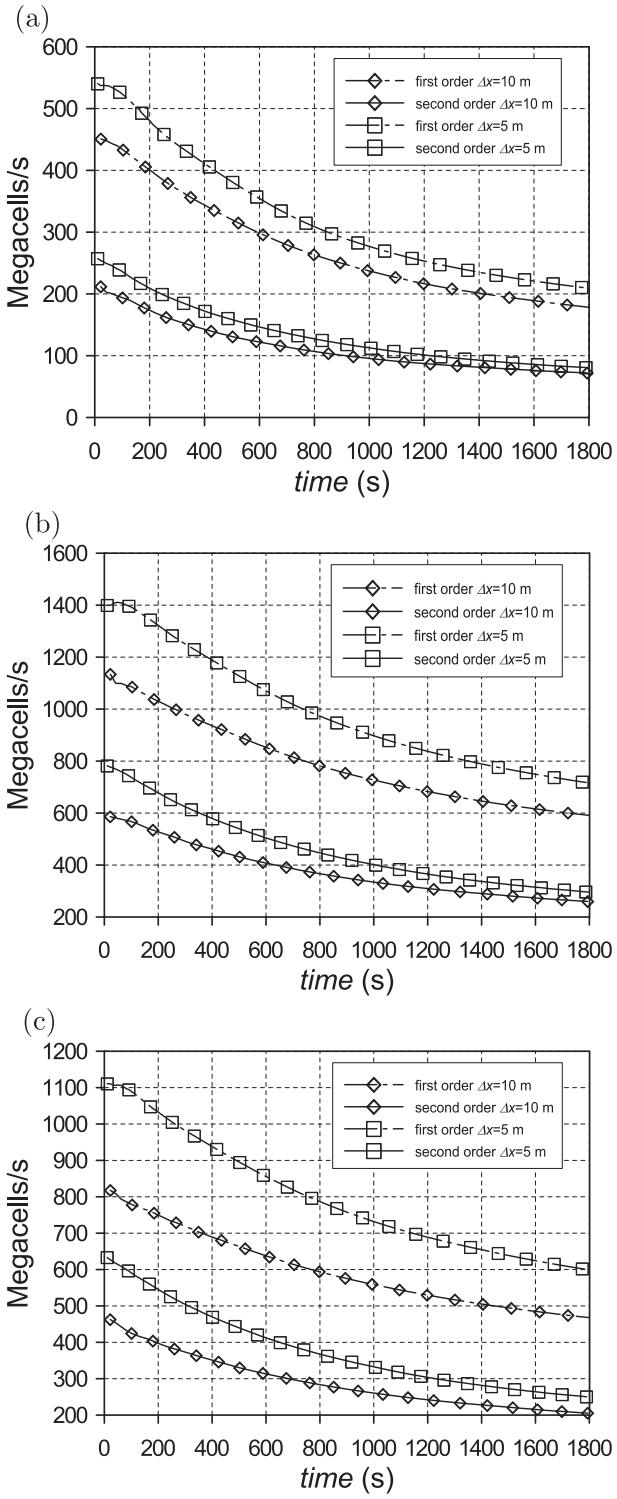

Fig. 13. Parma flood control reservoir dam-break: Megacells/s obtained with (a) GTS 450, (b) GTX 580 and (c) Tesla M2070.

The simulation has been conducted considering two different GPUs, three different grid sizes $\Delta x = 0.01$, 0.005 and 0.0025 m and with a wet-dry threshold $h_\varepsilon = 5 \times 10^{-5}$ m. Fig. 10 shows the water depth at points with coordinates $[x, y] = [3.00, 2.00]$ m, [2.50, 2.00] m and [1.60, 2.00] m obtained using a grid with size $\Delta x = 0.01$ m. The first point gets wet and dry during the periodic motion, the second gets dry only at $t = (0.25 + n)T$, $(n \in N)$ where $T = 1/\omega$ is the period, whereas the third remains wet all the time. In

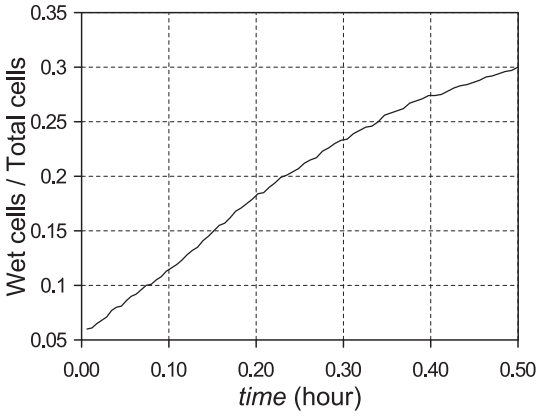

Fig. 14. Parma flood control reservoir dam-break: ratio between number of dry and total cells.

Table 4
Parma flood control reservoir dam-break: (physical time)/(computational time) ratio for first order (FO) and second order (SO) model.

	GTS 450		GTX 580		Tesla M2070	
Δx	FO	SO	FO	SO	FO	SO
10 m	107	43	338	152	261	119
5 m	16	6	53	28	43	18

all points there is an almost perfect agreement with the analytical solution.

Fig. 11 shows the water elevation maps at different timesteps: the shoreline remains circular during the simulation and the water surface elevation remains planar. This means that the numerical scheme is able to accurately simulate wet/dry fronts.

Table 3 compares the Megacells/s obtained using two different GPUs. The Megacells/s obtained with the serial version of the code using an Intel Xeon CPU X5365 3.00 GHz, a second order accurate model and $\Delta x = 0.01$ m, is equal to 1.3. Thus, the speedup obtained for the present test case varies from 106 (for the Tesla M2070) to 130 (for the GTX 580 GPU). In this example, the number of wet cells is roughly constant during the simulation, while the performances in terms of MC/s increase for finer meshes, regardless of the GPU in use and the order accuracy. This can be justified by the fact that, in the presence of finer meshes, a higher number of active blocks occupy the GPU scheduler and cores for longer time. Therefore, the heavier use of GPU time balances the CPU–GPU communication overhead. It is important to recall that the increase of MC/s values

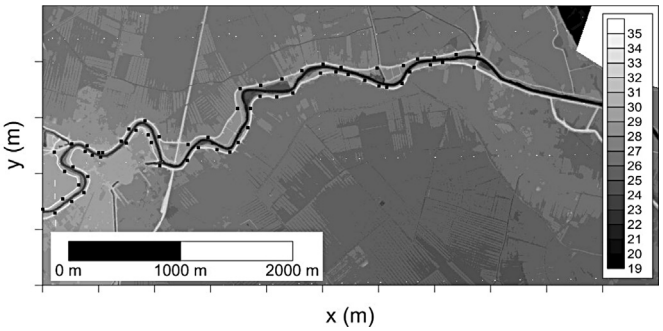

Fig. 15. Bathymetry of the Parma river reach. Black points represent the locations where maximum water surface elevation on left and right dykes were recorded.

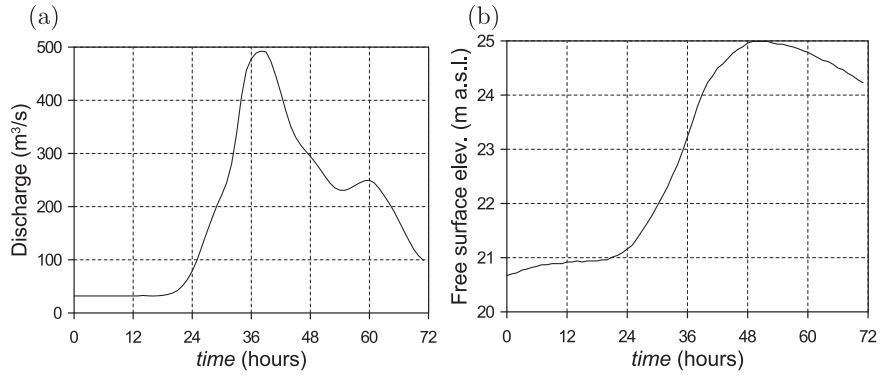

Fig. 16. January 2009 Parma river flood event: (a) inflow and (b) outflow boundary conditions.

for finer meshes does not translate into faster simulation times, since each increased mesh contains more cells to be processed.

4.6. Parma flood control reservoir dam-break

In order to test the numerical scheme to real cases characterized by a highly irregular topography, the flooding after the hypothetical collapse of the barrage on Parma river (Northern Italy) is simulated. The dam, built in 2005 for flood protection, is located about 10 km upstream from the city of Parma (Italy). The reservoir has a storage capacity of about $12 \cdot 10^6$ m^3 and the maximum water depth, with reference to the bottom of the stilling basin, is 16.4 m. At the initial time, the static condition corresponding to the maximum retaining depth (105.6 m a.s.l.) was imposed in the reservoir. The Manning roughness coefficient was set at 0.03 s m$^{-1/3}$ in the river and 0.05 s m$^{-1/3}$ elsewhere; the wet/dry tolerance h_ε was set at 10^{-3} m and two different simulations were run with grid sizes $\Delta x = 10$ and 5 m respectively. The domain is 5660 m long and 6150 m wide. The simulation was stopped after 30 min, when the reservoir is almost completely empty.

The water surface elevation at times 2, 10 and 30 min is shown in Fig. 12. The results are obtained with the first order numerical scheme and with a grid size $\Delta x = 10$ m, which was also the original resolution of the Digital Elevation Model (DEM) data. The results obtained with the second order numerical scheme are substantially analogous.

The Megacells/s obtained using three different GPUs are shown in Fig. 13. As reported in Fig. 14, the ratio between the number of wet cells and the total number of cells is equal to 6% and 30% at the beginning and at the end of the simulation, respectively. Due to the Block Deactivation Optimization, the efficiency is remarkably increased especially at early times, where the number of dry blocks is higher. For example, let us consider the Tesla M2070 GPU and a first order model with $\Delta x = 5$ m. At the beginning of the simulation the code performs at about 1100 Megacells/s which decreases down to approximately 600 Megacells/s at the end of the simulation. This means that the code is more efficient at the beginning of the simulation, when the number of dry cells is higher. This is due to the fact that the BDO procedure is able to almost double the efficiency of the numerical scheme. For these test cases the speedup over a serial CPU code varies from 70 to 200, depending on the CPU and GPU considered. The ratio between the physical and computational time γ is reported in Table 4. Considering the grid with size $\Delta x = 10$ m, and the GTX 580 GPU, γ is equal to 338 and 152 for the first and second order of accuracy, respectively, and thus the model can be used for fast simulations of real floods.

4.7. January 2009 Parma river flood

In this test case the flood which occurred in the Parma river in January 2009 was simulated. We considered the portion of the Parma river included between the city of Colorno and the confluence in the Po river. The bathymetry of the domain is plotted in Fig. 15. As upstream and downstream boundary conditions, the inflow discharge hydrograph (Fig. 16-a) and the water level hydrograph at the Po river confluence (Fig. 16-b) are respectively imposed. For this real flood event the maximum water surface elevation at the left and right dykes were acquired after the events at the points plotted in Fig. 15 by the Po river Agency (AIPo).

As shown in Fig. 15, the portion of the Parma river simulated is about 8 km long and it is included in a rectangular domain with dimensions 5654 × 1458 m. The simulation is run using a Cartesian

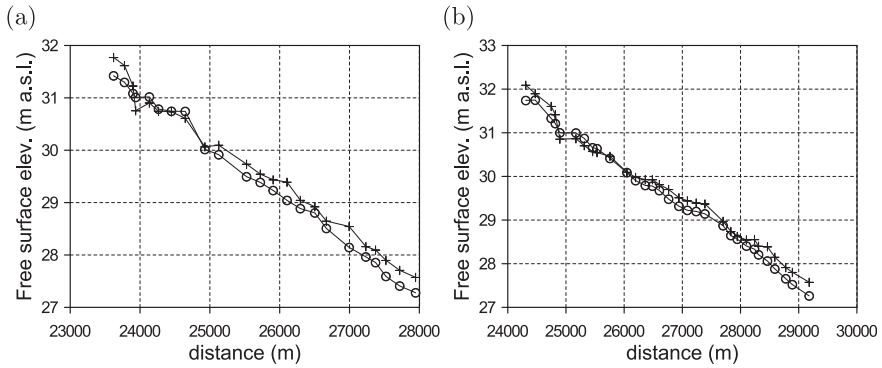

Fig. 17. January 2009 Parma river flood event: comparison between the registered and simulated maximum water surface elevation (a) on the left and (b) on the right dykes.

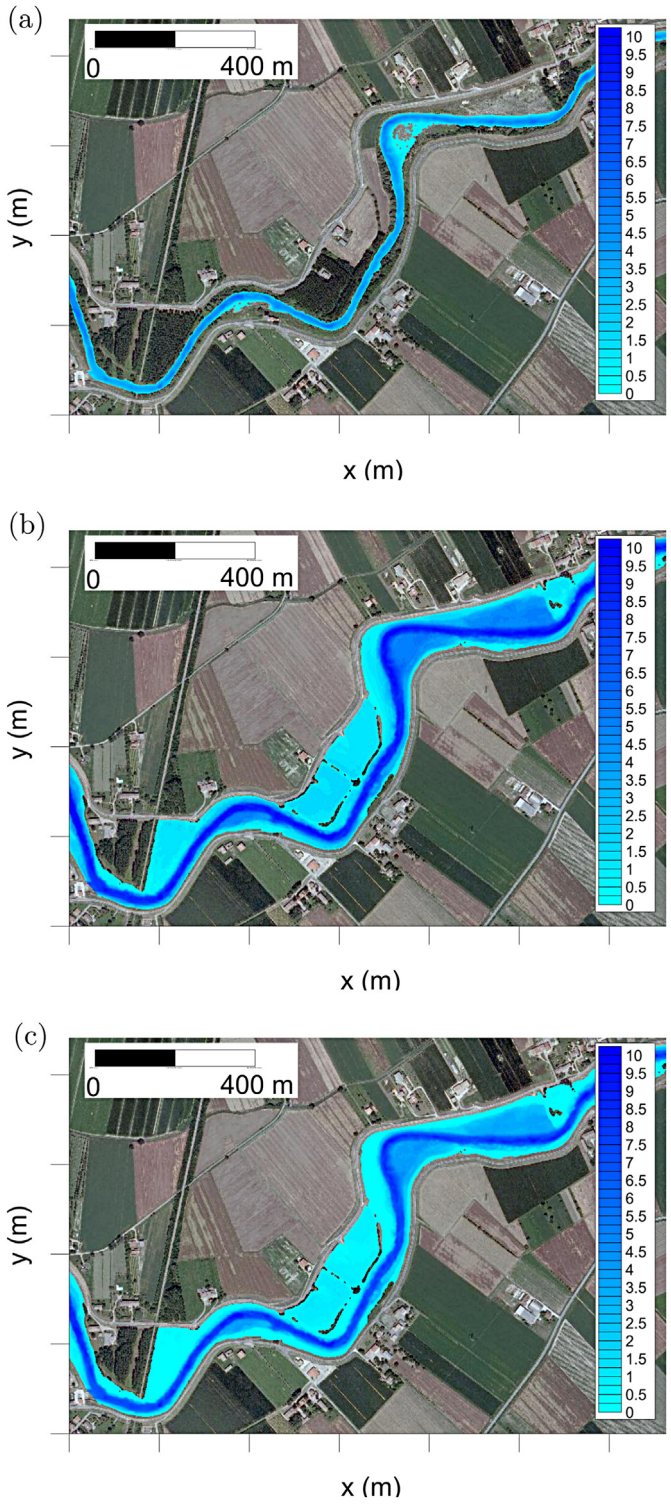

Fig. 18. January 2009 Parma river flood event: zoom of the depth maps at (a) 24 (b) 36 and (c) 48 h.

Table 5
January 2009 Parma river Flood event: Megacells/s for First Order (FO) and Second Order (SO) schemes.

	GTX 580		Tesla M2070	
Δx	FO	SO	FO	SO
2 m	1090	612	740	460

grid with size $\Delta x = 2$ m with about 2×10^6 cells, the wet-dry threshold h_ε is set to 2×10^{-4} m. In order to reproduce the registered water level profiles the Manning roughness coefficient has been set equal to 0.04 s $m^{-1/3}$ in the main channel and 0.067 s $m^{-1/3}$ in the floodplains. The comparisons between the registered and simulated water level profiles at the left and right levees are shown in Fig. 17, whereas the water depth at different timesteps is shown in Fig. 18. The numerical model is able to simulate a real event with complex inflow/outflow boundary conditions obtaining good agreement with the registered data.

In Table 5 the Megacells/s obtained using two different GPU and first and second order models are reported. Since the runtime of this test case on a serial code is too high (the order of months) no speedup comparison is possible. The runtime on a Tesla M2070 GPU, using a second order accurate scheme is equal to 4.03 h, with a ratio between the physical time and the runtime γ equal to 18. With the GTX 580, the computational time for the second order accurate simulation is equal to 3.03 h with $\gamma = 24$. Therefore also for this simulation with large number of cells (>2 Million cells), the GTX GPU is more efficient than the Tesla. This is not surprising, since, as pointed out at the beginning of the section, the GTX card is equipped with more and faster cores than the Tesla. The latter can be used in case of larger simulations, where the memory requirements overpass the capability of the GTX card. The performances for this test case are enhanced by the BDO procedure, since a large part of computation is skipped due to the fact that about the 90% of the domain remains dry during the simulation. Moreover, the absolute number of active blocks is large and it allows the GPU to fully occupy its multiprocessors (with a good balancing of launching kernel overheads). Finally, the typical active block (e.g., in the middle of the domain) contains no boundary conditions, which allows the most symmetrical processing on the GPU and therefore the best scalable result. This test case required 159 and 359 MB of GPU memory, for first and second order simulations respectively, which can be easily accommodated on recent GPUs.

5. Conclusions

In this paper a parallel implementation of a first and second order Shallow Water numerical scheme has been developed using the Compute Unified Device Architecture framework available for NVIDIA™ Graphics Processor Units. The algorithm features a high accuracy and a robust treatment of partially wet domains with moving wet/dry fronts. The difficulty in providing an efficient implementation for a GPU card of moving wet/dry front and arbitrarily complex open and closed boundary conditions have been addressed in this work.

Several optimization procedures, including the Block Deactivation Optimization, the implicit local ghost approach with memorization of border information (see Section 3.4) and a careful implementation to reduce the information exchanges between CPU and GPU, led to speedups of two order of magnitude in comparison with serial CPU codes for several different test cases herein simulated. In practice, simulations with several millions of cells and high ratios between the physical and the computational time can be conducted on a personal computer equipped with a CUDA-enabled GPU card. The presented system allows to perform high resolution simulations of large domains by solving the 2-D Shallow Water Equations, avoiding the use of simplified models (either with low resolution or using simplified Shallow Water Equations), that might produce less accurate results.

In order to obtain analogous performances, the current state-of-the-art is based on the use of large cluster machines, which are expensive and need continuous maintenance. We believe that our study takes advantage of the new opportunities offered by GPU

devices. In particular, the fast-developing hardware showed a growth rate in terms of capabilities (e.g. GFLOPS), which is way beyond the one for a standard CPU chip. Moreover, the cost per GPU multiprocessor is quickly decreasing, as opposed to the corresponding CPU cost rate. In this rapidly evolving scenario, it is expected that numerical application will benefit from the capability of running the code on GPU and/or cluster of GPUs, rather than traditional clusters of CPUs. Our study is aimed at exploiting the GPU hardware capabilities and the paper showed a number of successful test cases. Since the ratio between physical and computational time is always high, the numerical scheme herein presented can be embedded in real-time simulation tools, which can provide accurate and fast predictions and being useful also for flood management of occurring flood events.

In the paper, we experimentally tested different GPU devices and pointed out how a HPC card such as Tesla M2070 did not deliver top performances in terms of simulation speed, mainly because its architecture is designed for data intensive elaborations. In fact, a cheaper GTX 580 GPU allowed faster simulations, given its higher number of processors and clock speed. However, for real case simulations with larger grid sizes, the Tesla card would be the only option.

As future work, we plan to investigate how to allow larger domains processing, both with multi-scale approaches and cluster of GPUs.

Acknowledgment

The authors are grateful to Dr A. Maranzoni and Dr. F. Aureli for providing the original version of the serial Finite Volume scheme and for their fruitful comments.

Appendix A. Depth-positive MUSCL reconstruction

In the second order numerical scheme left and right state for the HLLC Riemann solver are calculated using a Monotone Upstream-centered Scheme for Conservation Laws (MUSCL) reconstruction. In the x-direction the interfaces for the generic cell i are indicated with the subscript $i \pm ½$. The generic conserved variable ψ is reconstructed at the left (superscript L) and right side (superscript R) as follows:

$$\begin{aligned}\psi^L_{i+\frac{1}{2},j} &= \psi_{i,j} + \frac{1}{2}\phi_{i,j}\left(\psi_{i,j} - \psi_{i-1,j}\right),\\ \psi^R_{i-\frac{1}{2},j} &= \psi_{i,j} - \frac{1}{2}\phi_{i,j}\left(\psi_{i+1,j} - \psi_{i,j}\right)\end{aligned} \tag{A.1}$$

where $\phi_{i,j}$ is the minmod limiter function (Toro, 1999a,b).

To avoid any negative depth generated by the high order scheme, the positive preserving reconstruction of Audusse et al. (2004) is herein adopted. η, h, (uh) and (vh) are reconstructed using Equation (A.1), afterward the bed elevation at $i \pm ½$ is obtained as follows:

$$\begin{aligned}z^L_{i+\frac{1}{2},j} &= \eta^L_{i+\frac{1}{2},j} - h^L_{i+\frac{1}{2},j},\\ z^R_{i-\frac{1}{2},j} &= \eta^R_{i-\frac{1}{2},j} - h^R_{i-\frac{1}{2},j}\end{aligned} \tag{A.2}$$

then the provisional velocity are calculated as:

$$\begin{aligned}u^L_{i+\frac{1}{2},j} &= \frac{(uh)^L_{i+\frac{1}{2},j}}{h^L_{i+\frac{1}{2},j}} \quad u^R_{i-\frac{1}{2},j} = \frac{(uh)^R_{i-\frac{1}{2},j}}{h^R_{i-\frac{1}{2},j}}\\ v^L_{i+\frac{1}{2},j} &= \frac{(vh)^L_{i+\frac{1}{2},j}}{h^L_{i+\frac{1}{2},j}} \quad v^R_{i-\frac{1}{2},j} = \frac{(vh)^R_{i-\frac{1}{2},j}}{h^R_{i-\frac{1}{2},j}}\end{aligned} \tag{A.3}$$

the left and right values of the bottom elevation are then used to define a single value bottom elevation as:

$$\begin{aligned}z_{i+\frac{1}{2},j} &= \max\left[z^L_{i+\frac{1}{2},j}, z^R_{i+\frac{1}{2},j}\right]\\ z_{i-\frac{1}{2},j} &= \max\left[z^L_{i-\frac{1}{2},j}, z^R_{i-\frac{1}{2},j}\right]\end{aligned} \tag{A.4}$$

then the water depth is reconstructed as:

$$\begin{aligned}\overline{h}^L_{i+\frac{1}{2},j} &= \max\left[0, \eta^L_{i+\frac{1}{2},j} - z_{i+\frac{1}{2},j}\right]\\ \overline{h}^R_{i-\frac{1}{2},j} &= \max\left[0, \eta^R_{i-\frac{1}{2},j} - z_{i-\frac{1}{2},j}\right]\end{aligned} \tag{A.5}$$

This clearly ensures that the reconstructed water depths are always positive. The Riemann states of water surface elevation and the specific discharges are recovered as follows:

$$\begin{aligned}\overline{\eta}^L_{i+\frac{1}{2},j} &= \overline{h}^L_{i+\frac{1}{2},j} + z_{i+\frac{1}{2},j}\\ \overline{\eta}^R_{i-\frac{1}{2},j} &= \overline{h}^R_{i-\frac{1}{2},j} + z_{i-\frac{1}{2},j}\end{aligned} \tag{A.6}$$

$$\begin{aligned}\overline{(uh)}^L_{i+\frac{1}{2},j} &= \overline{h}^L_{i+\frac{1}{2},j} u^L_{i+\frac{1}{2},j} \quad \overline{(uh)}^R_{i-\frac{1}{2},j} = \overline{h}^R_{i-\frac{1}{2},j} u^R_{i-\frac{1}{2},j}\\ \overline{(vh)}^L_{i+\frac{1}{2},j} &= \overline{h}^L_{i+\frac{1}{2},j} v^L_{i+\frac{1}{2},j} \quad \overline{(vh)}^R_{i-\frac{1}{2},j} = \overline{h}^R_{i-\frac{1}{2},j} v^R_{i-\frac{1}{2},j}\end{aligned} \tag{A.7}$$

The MUSCL reconstruction is analogous in y-direction.

Appendix B. Bed slope source term discretization

If a second order numerical scheme is adopted, to obtain a well-balance numerical scheme the slope source term discretization needs to be consistent with the MUSCL reconstruction described in the previous section. The slope source term in x direction S_x is discretized as follows (Liang and Marche, 2009):

$$S_x = -g\frac{\overline{\eta}^R_{i-\frac{1}{2},j} + \overline{\eta}^L_{i+\frac{1}{2},j}}{2}\left[\frac{z_{i+\frac{1}{2},j} - z_{i-\frac{1}{2},j}}{\Delta x}\right] + S^- + S^+ \tag{B.1}$$

S^- and S^+ are defined as

$$\begin{aligned}S^- &= g\Delta z_{i-\frac{1}{2},j}\frac{z_{i+\frac{1}{2},j} - z_{i-\frac{1}{2},j} + \Delta z_{i-\frac{1}{2},j}}{2\Delta x}\\ S^+ &= g\Delta z_{i+\frac{1}{2},j}\frac{z_{i+\frac{1}{2},j} - z_{i-\frac{1}{2},j} + \Delta z_{i+\frac{1}{2},j}}{2\Delta x}\end{aligned} \tag{B.2}$$

where:

$$\begin{aligned}\Delta z_{i-\frac{1}{2},j} &= \max\left[0, z_{i-\frac{1}{2},j} - \eta^R_{i-\frac{1}{2},j}\right] \quad \Delta z_{i+\frac{1}{2},j}\\ &= \max\left[0, z_{i+\frac{1}{2},j} - \eta^L_{i+\frac{1}{2},j}\right]\end{aligned}$$

the two corrective terms $\Delta z_{i-½,j}$ and $\Delta z_{i+½,j}$ guarantee that the C-property is preserved also in presence of wet-dry fronts. If a first order numerical scheme is used, then the slope source term is discretized with the same formulation but the bed elevation at $i \pm ½$ is obtained as follows:

$$\begin{aligned}z^L_{i+\frac{1}{2},j} &= \max\left[z_{i+1,j}, z_{i,j}\right]\\ z^R_{i-\frac{1}{2},j} &= \max\left[z_{i-1,j}, z_{i,j}\right]\end{aligned} \tag{B.3}$$

and the free surface elevation values in Equation (B.1) are simply the values in the cells with no reconstruction: $\overline{\eta}^{R}_{i-1/2,\,j} = \overline{\eta}^{L}_{i+1/2,\,j} = \eta_{i,\,j}$.

The source term discretization of Equation (B.1) together with the positive-depth reconstruction described in the previous section create a well-balance discretization of the SWEs as demonstrated by Liang and Marche (2009).

References

Akbar, M., Aliabadi, S., 2013. Hybrid numerical methods to solve shallow water equations for hurricane induced storm surge modeling. Environ. Model. Softw. 46, 118–128.

Audusse, E., Bouchut, F., Bristeau, M., Klein, R., Perthame, B., 2004. A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows. SIAM J. Sci. Comput. 25, 2050–2065.

Aureli, F., Maranzoni, A., Mignosa, P., Ziveri, C., 2008. A weighted surface-depth gradient method for the numerical integration of the 2d shallow water equations with topography. Adv. Water Resour. 31, 962–974.

Balzano, A., 1998. Evaluation of methods for numerical simulation of wetting and drying in shallow water flow models. Coast. Eng. 34, 83–107.

Bates, P., Hervouet, J., 1999. A new method for moving boundary hydrodynamic problems in shallow water. Proc. R. Soc. Lond. A 455, 3107–3128.

Bates, P., Horritt, M., 2005. Modelling wetting and drying processes in hydraulic models. In: Bates, S.L.P.D., Ferguson, R. (Eds.), Computational Fluid Dynamics: Applications in Environmental Hydraulics. John Wiley & Sons, pp. 121–146.

Bates, P., Roo, A.D., 2000. A simple raster-based model for flood inundation simulation. J. Hydrol. 236, 54–77.

Begnudelli, L., Sanders, B., 2006. Unstructured grid finite-volume algorithm for shallow-water flow and scalar transport with wetting and drying. ASCE J. Hydr. Eng. 132, 371–384.

Bermúdez, A., Vázquez, M., 1994. Upwind methods for hyperbolic conservation laws with source terms. Comput. Fluids 23, 1049–1071.

Bermúdez, A., Dervieux, A., Desideri, J., Vázquez, M., 1998. Upwind schemes for the two-dimensional shallow water equations with variable depth using unstructured meshes. Comput. Methods Appl. Mech. Eng. 155, 49–72.

Bradford, S., Sanders, B., 2002. Finite-volume model for shallow-water flooding of arbitrary topography. J. Hydraul. Eng. 128, 289–298.

Brodtkorb, A.R., Saetra, M.L., Altinakar, M., 2012. Efficient shallow water simulations on gpus: implementation, visualization, verification, and validation. Comput. Fluids 55, 1–12.

Brodtkorb, A.R., Hagen, T.R., Saetra, M.L., 2013. Graphics processing unit (gpu) programming strategies and trends in {GPU} computing. J. Parallel Distrib. Comput. 73, 4–13. ¡ce:title¿Metaheuristics on GPUs¡/ce:title¿.

Caleffi, V., Valiani, A., Zanni, A., 2003. Finite volume method for simulating extreme flood events in natural channels. J. Hydraul. Res. 41, 167–177.

Castro, M.J., Ortega, S., de la Asunción, M., Mantas, J.M., Gallardo, J.M., 2011. Gpu computing for shallow water flow simulation based on finite volume schemes. C. R. Mec. 339, 165–184.

Casulli, V., 1990. Semi-implicit finite difference methods for the two-dimensional shallow water equations. J. Comput. Phys. 86, 56–74.

Crespo, A.C., Dominguez, J.M., Barreiro, A., Gmez-Gesteira, M., Rogers, B.D., 2011. Gpus, a new tool of acceleration in cfd: efficiency and reliability on smoothed particle hydrodynamics methods. PLoS One 6, e20685.

Cuda best practices guide, docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

de la Asunción, M., Mantas, J.M., Castro, M.J., 2011. Simulation of one-layer shallow water systems on multicore and cuda architectures. J. Supercomput. 58, 206–214.

de la Asunción, M., Castro, M.J., Fernández-Nieto, E., Mantas, J.M., Acosta, S.O., González-Vida, J.M., 2013. Efficient gpu implementation of a two waves tvd-waf method for the two-dimensional one layer shallow water system on structured meshes. Comput. Fluids 80, 441–452.

de la Asunción, M., Mantas, J.M., Castro, M.J., Fernández-Nieto, E., 2012. An MPI-CUDA implementation of an improved roe method for two-layer shallow water systems. J. Parallel Distrib. Comput. 72 (9), 1065–1072.

Defina, A., 2000. Two dimensional shallow flows equations for partially dry areas. Water Resour. Res. 36, 3251–3264.

Fernando, R., Kilgard, M.J., 2003. The CG Tutorial: the Definitive Guide to Programmable Real-time Graphics. Addison Wesley Pub Co Inc.

Feyen, L., Watkiss, P., 2011. The Climatecost Project. final report. Stockholm Environment Institute, Sweden.

Gallardo, J.M., Ortega, S., Asunción, M., Mantas, J.M., 2011. Two-dimensional compact third-order polynomial reconstructions. solving nonconservative hyperbolic systems using gpus. J. Sci. Comput. 48, 141–163.

García-Navarro, P., Vázquez, M., 2000. On numerical treatment of the source terms in shallow water equations. Comput. Fluids 29, 951–979.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V., 2008. Parallel computing experiences with cuda. Micro, IEEE 28, 13–27.

Greenberg, J., Leroux, A., 1996. A well-balanced scheme for the numerical processing of source terms in hyperbolic equations. SIAM J. Numer. Anal. 33, 1–16.

Hubbard, M., García-Navarro, P., 2000. Flux difference splitting and the balancing of source terms and flux gradients. J. Comp. Phys. 165, 89–125.

Kalyanapu, A.J., Shankar, S., Pardyjak, E.R., Judi, D.R., Burian, S.J., 2011. Assessment of {GPU} computational enhancement to a 2d flood model. Environ. Model. Softw. 26, 1009–1016.

Kirk, D.B., Hwu, W., 2010. Programming Massively Parallel Processors. A Hands-on Approach. Morgan Kaufmann/Elsevier.

Lamb, R., Crossley, M., Waller, S., 2009. A fast two-dimensional floodplain inundation model. Proc. ICE Water Manag. 162, 363–370.

Lastra, M., Mantas, J.M., Ureña, C., Castro, M.J., García-Rodríguez, J.A., 2009. Simulation of shallow-water systems using graphics processing units. Math. Comput. Simul. 80, 598–618.

LeVeque, R., 1998. Balancing source terms and flux gradients in high-resolution methods: the quasi-steady wave-propagation algorithms. J. Comp. Phys. 146, 346–365.

LeVeque, R.J., 2002. Finite Volume Methods for Hyperbolic Problems. Cambridge University Press.

Liang, Q., Borthwick, A.G.L., 2009. Adaptive quadtree simulation of shallow flows with wet-dry fronts over complex topography. Comput. Fluids 38, 221–234.

Liang, Q., Marche, F., 2009. Numerical resolution of well-balanced shallow water equations with complex source terms. Adv. Water Resour. 32, 873–884.

Miglio, E., Quarteroni, A., Saleri, F., 1999. Finite element approximation of quasi-3d shallow water equations. Comput. Methods Appl. Mech. Eng. 174, 355–369.

Neal, J.C., Fewtrell, T.J., Bates, P.D., Wright, N.G., 2010. A comparison of three parallelisation methods for 2d flood inundation models. Environ. Model. Softw. 25, 398–411.

Neal, J., Villanueva, I., Wright, N., Willis, T., Fewtrell, T., Bates, P., 2012. How much physical complexity is needed to model flood inundation? Hydrol. Process. 26, 2264–2282.

Néelz, S., Pender, G., 2010. Benchmarking of 2d Hydraulic Modelling Packages. Environment Agency, Bristol, UK.

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J., 2008. Gpu computing. Proc. IEEE 96, 879–899.

Rogers, B.D., Borthwick, A.G.L., Taylor, P.H., 2003. Mathematical balancing of flux gradient and source terms prior to using roes approximate Riemann solver. J. Comput. Phys. 192, 422–451.

Saetra, M.L., Brodtkorb, A.R., 2012. Shallow Water Simulations on Multiple GPUs. In: Lecture Notes in Computer Science, vol. 7134, pp. 56–66.

Sanders, B.F., Schubert, J.E., Detwiler, R.L., 2010. Parbrezo: a parallel, unstructured grid, godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale. Adv. Water Resour. 33, 1456–1467.

Soares-Frazo, S., Guinot, V., Lhomme, J., Zech, Y., 2007. Conservative discretization of bed slope source terms on irregular topographies. In: Proceedings of 32nd Congress of IAHR, Venice, Italy.

Thacker, W.C., 1981. Some exact solutions to the nonlinear shallow-water wave equations. J. Fluid Mech. 107, 499–508.

Toro, E., 1999a. Shock Capturing Methods for Free Surface Shallow Water Flows. Wiley, New York.

Toro, E., 1999b. Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer.

Vacondio, R., Rogers, B.D., Stansby, P.K., 2012. Accurate particle splitting for smoothed particle hydrodynamics in shallow water with shock capturing. Int. J. Numer. Methods Fluids 69, 1377–1410.

Valiani, A., Begnudelli, L., 2006a. Divergence form for the bed slope source term in shallow water equations. ASCE J. Hydraul. Eng. 146, 652–665.

Valiani, A., Begnudelli, L., 2006b. Divergence form for bed slope source term in shallow water equations. J. Hydraul. Eng. 132, 652–665.

Vázquez, M., 1999. Improved treatment of source terms in upwind schemes for the shallow water equations in channels with irregular geometry. J. Comp. Phys. 32, 101–136.

Zhou, J., Causon, D., Mingham, C., Ingram, D., 2001. The surface gradient method for the treatment of source terms in the shallow-water equations. J. Comput. Phys. 168, 1–25.