C. R. Mecanique 339 (2011) 165–184

Contents lists available at ScienceDirect

Comptes Rendus Mecanique

www.sciencedirect.com

High Performance Computing / Le Calcul Intensif

GPU computing for shallow water flow simulation based on finite volume schemes

Manuel J. Castro [a,*], Sergio Ortega [a], Marc de la Asunción [b], José M. Mantas [b], José M. Gallardo [a]

[a] Dpto. de Análisis Matemático, Universidad de Málaga, Campus de Teatinos s/n, 29080 Malaga, Spain
[b] Dpto. de Lenguajes y Sistemas Informáticos, Universidad de Granada, C./Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain

ARTICLE INFO

Article history:
Available online 30 December 2010

Keywords:
Computer science
GPUs
Finite volume methods
Shallow water
High-order schemes

ABSTRACT

This article is a review of the work that we are carrying out to efficiently simulate shallow water flows. In this paper, we focus on the efficient implementation of path-conservative Roe type high-order finite volume schemes to simulate shallow flows that are supposed to be governed by the one-layer or two-layer shallow water systems, formulated under the form of a conservation law with source terms. The implementation of the scheme is carried out on Graphics Processing Units (GPUs), thus achieving a substantial improvement of the speedup with respect to normal CPUs. Finally, some numerical experiments are presented.

1. Introduction

This article deals with the development of efficient GPU implementations of Finite Volume solvers on structured and unstructured grids to solve 2d hyperbolic systems of conservation laws with source terms and nonconservative products.

Problems of this nature arise, for example, in Computational Fluid Dynamics. We are concerned in particular with the simulation of free-surface waves in shallow layers of homogeneous fluids or internal waves in stratified fluids composed by two shallow layers of immiscible liquids. The motion of a layer of homogeneous fluid is supposed here to be governed by the shallow water system, formulated under the form of a conservation law with source terms or *balance law*. In the stratified case, the flow is supposed to be governed by a system composed by two coupled shallow water systems. The global system can be formulated under the form of two coupled balance laws, the coupling terms having the form of nonconservative products.

We are mainly interested in the application of these systems to geophysical flows: models based on shallow water systems are useful for the simulation of rivers, channels, dambreak problems, ocean currents, estuarine systems, etc. Simulating those phenomena gives place to very long lasting simulations in big computational domains, so extremely efficient implementations are needed to be able to analyze those problems in low computational time. Moreover, tsunami propagation or floods could require real time calculation.

On the other hand, the numerical schemes to solve these equations usually exhibit a high degree of potential data parallelism. These facts suggest the design of parallel versions of the numerical schemes for parallel machines in order to solve and analyze these problems in reasonable execution times.

In this article, we tackle the acceleration of several finite volume numerical schemes to solve one and two-layer shallow water systems.

* Corresponding author.
E-mail addresses: castro@anamat.cie.uma.es (M.J. Castro), sergio@anamat.cie.uma.es (S. Ortega), marc@correo.ugr.es (M. de la Asunción), jmmantas@ugr.es (J.M. Mantas), gallardo@anamat.cie.uma.es (J.M. Gallardo).

1631-0721/$ – see front matter
doi:10.1016/j.crme.2010.12.004

An interesting first order finite volume scheme to solve shallow water systems [1] has been parallelized and optimized by combining a distributed implementation which runs on a PC cluster [1] with the use of SSE-optimized routines [2]. However, despite of the important performance improvements, a greater reduction of the runtimes is necessary in order to use these schemes in realistic applications.

A cost effective way of obtaining a substantially higher performance in these applications consists in using the modern Graphics Processor Units (GPUs). The use of these devices to accelerate computationally intensive tasks is growing in popularity among the scientific and engineering community [3,4]. Modern GPUs present a massively parallel architecture which includes hundreds of processing units optimized for performing floating point operations and multithreaded execution. These architectures make it possible to obtain performances that are orders of magnitude faster than a standard CPU at a very affordable price.

There are previous proposals to implement finite volume one-layer shallow water solvers on GPUs by using a graphics-specific programming language [5–7]. These solvers obtain considerable speedups to simulate one-layer shallow water system but their graphics-based design is not easy to understand and maintain.

Recently, NVIDIA has developed the CUDA programming toolkit [8] which includes an extension of the C language and facilitates the programming of GPUs for general purpose applications by preventing the programmer to deal with the graphics details of the GPU.

A CUDA solver for one-layer system based on the first order finite volume scheme presented in [1] is described in [9] to deal with structured regular meshes. The extension of this CUDA solver to deal with two-layer shallow water system is presented in [10]. There also exists proposals to implement high-order schemes to simulate one-layer system using CUDA-enabled GPUs [11,12].

The structure of the paper is as follows: in Section 2, the one-layer and two-layer shallow water system are introduced. Next, some comments about the difficulty of the definition of weak solutions for those systems are given. In Section 4, the high-order finite volume scheme developed in [13] is reviewed. Next, some details about the implementation on GPUs of the numerical schemes for structured and unstructured meshes are presented as well as some numerical experiments. Finally, some conclusions are drawn in Section 8.

2. Equations

We consider a general problem consisting of a system of conservation laws with nonconservative products and source terms given by:

$$\frac{\partial U}{\partial t} + \frac{\partial F_1}{\partial x}(U) + \frac{\partial F_2}{\partial y}(U) = B_1(U)\cdot\frac{\partial U}{\partial x} + B_2(U)\cdot\frac{\partial U}{\partial y} + S_1(U)\frac{\partial H}{\partial x} + S_2(U)\frac{\partial H}{\partial y} \tag{1}$$

where $U(\boldsymbol{x},t): D\times(0,T)\mapsto \Omega\subset\mathbb{R}^N$, being D a bounded domain of $\mathbb{R}^2$; $\boldsymbol{x}=(x,y)$ denotes an arbitrary point of D; Ω is an open convex subset of $\mathbb{R}^N$. Finally $F_i:\Omega\mapsto\mathbb{R}^N$, $B_i:\Omega\mapsto\mathcal{M}_N$, $S_i:\Omega\mapsto\mathbb{R}^N$, $i=1,2$, are regular functions, and $H: D\mapsto\mathbb{R}$ is a known function. Observe that if $B_1=B_2=0=S_1=S_2=0$, (1) is a system of conservation laws, and if $B_1=B_2=0$, (1) is a system of conservation laws with source term (or balance laws).

Notice that the nonconservative products $B_1(U)\partial_x U$, $B_2(U)\partial_y U$, $S_1(U)\partial_x H$, and $S_2(U)\partial_y H$ do not make sense in the sense of distributions for discontinuous solutions. The problem of giving a sense to the solution is a difficult task: we refer to [14].

Some examples of equations that fit into this abstract framework are the one-layer and two-layer shallow water systems.

(i) *One-layer shallow water system.*
The PDE system governing the flow of a shallow layer of fluid that occupies a subdomain $D\subset\mathbb{R}^2$ can be written in the form (1) with:

$$U=\begin{bmatrix} h\\ q_x\\ q_y\end{bmatrix} \tag{2}$$

$$F_1(U)=\begin{bmatrix} q_x\\ \dfrac{q_x^2}{h}+\dfrac{1}{2}gh^2\\ \dfrac{q_xq_y}{h}\end{bmatrix},\qquad F_2(U)=\begin{bmatrix} q_y\\ \dfrac{q_xq_y}{h}\\ \dfrac{q_y^2}{h}+\dfrac{1}{2}gh^2\end{bmatrix} \tag{3}$$

$$S_1(U)=\begin{bmatrix} 0\\ gh\\ 0\end{bmatrix},\qquad S_2(U)=\begin{bmatrix} 0\\ 0\\ gh\end{bmatrix} \tag{4}$$

where $H(\boldsymbol{x})$ is the depth function measured from a fixed level of reference; g is the gravity constant; $h(\boldsymbol{x},t)$ and $\boldsymbol{q}(\boldsymbol{x},t)=(q_x(\boldsymbol{x},t),q_y(\boldsymbol{x},t))$ are, respectively, the thickness and the mass-flow of the water layer at the point $\boldsymbol{x}$ at time t,

that are related to the velocity $\boldsymbol{u}(\boldsymbol{x},t)=(u_x(\boldsymbol{x},t),u_y(\boldsymbol{x},t))$ through the equality:

$$\boldsymbol{q}(\boldsymbol{x},t)=\boldsymbol{u}(\boldsymbol{x},t)h(\boldsymbol{x},t)$$

Here, $B_1=B_2=0$.

For the sake of simplicity friction terms are neglected.

(ii) *Two-layer shallow water system.*

The system of equations governing the 2d flow of two superposed immiscible layers of shallow fluids in a subdomain $D\subset\mathbb{R}^2$, can be also written in the form (1) with:

$$U=[h_1,\ q_{1,x},\ q_{1,y},\ h_2,\ q_{2,x},\ q_{2,y}]^T \tag{5}$$

$$F_1(U)=\begin{bmatrix} q_{1,1}\\ \dfrac{q_{1,x}^2}{h_1}+\dfrac{1}{2}gh_1^2\\ \dfrac{q_{1,x}q_{1,y}}{h_1}\\ q_{2,1}\\ \dfrac{q_{2,x}^2}{h_2}+\dfrac{1}{2}gh_2^2\\ \dfrac{q_{2,x}q_{2,y}}{h_2}\end{bmatrix},\qquad F_2(W)=\begin{bmatrix} q_{1,y}\\ \dfrac{q_{1,x}q_{1,y}}{h_1}\\ \dfrac{q_{1,y}^2}{h_1}+\dfrac{1}{2}gh_1^2\\ q_{2,y}\\ \dfrac{q_{2,x}q_{2,y}}{h_2}\\ \dfrac{q_{2,y}^2}{h_2}+\dfrac{1}{2}gh_2^2\end{bmatrix} \tag{6}$$

$$B_1(U)=\begin{bmatrix} 0&0&0&0&0&0\\ 0&0&0&-gh_1&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ -rgh_2&0&0&0&0&0\\ 0&0&0&0&0&0\end{bmatrix},\qquad B_2(U)=\begin{bmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&-gh_1&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ -rgh_2&0&0&0&0&0\end{bmatrix} \tag{7}$$

$$S_1(\boldsymbol{x},U)=[0,\ gh_1,\ 0,\ 0,\ gh_2,\ 0]^T \tag{8}$$

$$S_2(\boldsymbol{x},U)=[0,\ 0,\ gh_1,\ 0,\ 0,\ gh_2]^T \tag{9}$$

Index 1 makes reference to the upper layer and index 2 to the lower one. Again, g is the gravity and $H(\boldsymbol{x})$, the depth function measured from a fixed level of reference. $r=\frac{\rho_1}{\rho_2}$ is the ratio of the constant densities of the layers ($\rho_1<\rho_2$) which, in realistic oceanographical applications, is close to 1. Finally, $h_i(\boldsymbol{x},t)$ and $\boldsymbol{q}_i(\boldsymbol{x},t)$ are, respectively, the thickness and the mass-flow of the i-th layer at the point $\boldsymbol{x}$ at time t, and again they are related to the velocities $\boldsymbol{u}_i(\boldsymbol{x},t)=(u_{i,x}(\boldsymbol{x},t),u_{i,y}(\boldsymbol{x},t))$, $i=1,2$ by the equalities:

$$\boldsymbol{q}_i(\boldsymbol{x},t)=\boldsymbol{u}_i(\boldsymbol{x},t)h_i(\boldsymbol{x},t),\quad i=1,2$$

Again, friction terms are neglected.

Notice that system (1) can be rewritten as

$$W_t+\mathcal{A}_1(W)W_x+\mathcal{A}_2(W)W_y=0 \tag{10}$$

by considering $W=[U,H]^T$ and

$$\mathcal{A}_i(W)=\begin{pmatrix} J_i(U)-B_i(U) & -S_i(U)\\ 0 & 0\end{pmatrix},\quad i=1,2$$

where $J_i(U)=\frac{\partial F_i}{\partial U}(U)$, $i=1,2$ denote the Jacobians of the fluxes F_i, $i=1,2$.

3. Weak solutions

Let us consider system (10) where $W(\mathbf{x},t)$ takes values on a convex domain Ω of $\mathbb{R}^N$ and $\mathcal{A}_i$, $i=1,2$, are two smooth and locally bounded matrix-valued functions from Ω to $\mathcal{M}_{N\times N}(\mathbb{R})$.

We assume that (10) is strictly hyperbolic, i.e. for all $W\in\Omega$ and $\forall\boldsymbol{\eta}=(\eta_x,\eta_y)\in\mathbb{R}^2$, the matrix

$$\mathcal{A}(W,\boldsymbol{\eta})=\mathcal{A}_1(W)\eta_x+\mathcal{A}_2(W)\eta_y$$

has N real and distinct eigenvalues

$$\lambda_1(W,\boldsymbol{\eta}) < \cdots < \lambda_N(W,\boldsymbol{\eta})$$

$\mathcal{A}(W,\boldsymbol{\eta})$ is thus diagonalizable:

$$\mathcal{A}(W,\eta) = \mathcal{K}(W,\boldsymbol{\eta})\Lambda(W,\boldsymbol{\eta})\mathcal{K}^{-1}(W,\boldsymbol{\eta})$$

where $\Lambda(W,\boldsymbol{\eta})$ is the diagonal matrix whose coefficients are the eigenvalues of $\mathcal{A}(W,\boldsymbol{\eta})$ and $\mathcal{K}(W,\boldsymbol{\eta})$ is the matrix whose j-th column is an eigenvector $R_j(W,\boldsymbol{\eta})$ associated to the eigenvalue $\lambda_j(W,\boldsymbol{\eta})$, $j=1,\ldots,N$.

For discontinuous solutions W, the nonconservative products $\mathcal{A}_1(W)W_x$ and $\mathcal{A}_2(W)W_y$ do not make sense as distributions. However, the theory developed by Dal Maso, LeFloch and Murat in [14] allows to give a rigorous definition of nonconservative products as bounded measures provided that a family of Lipschitz continuous paths $\Psi\colon [0,1]\times\Omega\times\Omega\times\mathcal{S}^1\to\Omega$ is prescribed, where $\mathcal{S}^1\subset\mathbb{R}^2$ denotes the unit sphere. This family must satisfy certain natural regularity conditions, in particular:

(i) $\Psi(0;W_L,W_R,\boldsymbol{\eta}) = W_L$ and $\Psi(1;W_L,W_R,\boldsymbol{\eta}) = W_R$, for any $W_L, W_R\in\Omega$, $\boldsymbol{\eta}\in\mathcal{S}^1$.
(ii) $\Psi(s;W_L,W_R,\boldsymbol{\eta}) = \Psi(1-s;W_R,W_L,-\boldsymbol{\eta})$, for any $W_L, W_R\in\Omega$, $s\in[0,1]$, $\boldsymbol{\eta}\in\mathcal{S}^1$.

The choice of this family of paths should be based on the physics of the problem: for instance, it should be based on the viscous profiles corresponding to a regularized system in which some of the neglected terms (e.g. the viscous terms) are taken into account. Unfortunately, the explicit calculations of viscous profiles for a regularization of (10) is a difficult task. An alternative is to choose the 'canonical' choice given by the family of segments:

$$\Psi(s;W_L,W_R,\boldsymbol{\eta}) = W_L + s(W_R - W_L) \tag{11}$$

that corresponds to the definition of nonconservative products proposed by Volpert (see [15]).

Suppose that a family of paths Ψ in Ω has been chosen. Then a piecewise regular function W is a *weak solution* of (10) if and only if the two following conditions are satisfied:

(i) W is a classical solution where it is smooth.
(ii) At every point of a discontinuity W satisfies the jump condition

$$\int_0^1 \big(\sigma\mathcal{I} - \mathcal{A}\big(\Psi\big(s;W^-,W^+,\boldsymbol{\eta}\big),\boldsymbol{\eta}\big)\big)\frac{\partial\Psi}{\partial s}\big(s;W^-,W^+,\boldsymbol{\eta}\big)\,\mathrm{d}s = 0 \tag{12}$$

where $\mathcal{I}$ is the identity matrix; σ, the speed of propagation of the discontinuity; $\boldsymbol{\eta}$ a unit vector normal to the discontinuity at the considered point; and W^-, W^+, the lateral limits of the solution at the discontinuity.

As in conservative systems, together with the definition of weak solutions, a notion of entropy has to be chosen. We will assume here that the system can be endowed with an entropy pair $(\eta,\mathbf{G})$, i.e. a pair of regular functions $\eta:\Omega\to\mathbb{R}$ and $\mathbf{G}=(G_1,G_2):\Omega\to\mathbb{R}^2$ such that:

$$\nabla G_i(W) = \nabla\eta(W)\cdot\mathcal{A}_i(W),\quad \forall W\in\Omega,\ i=1,2$$

Then, a weak solution is said to be an *entropy solution* if it satisfies the inequality

$$\partial_t\eta(W) + \partial_x G_1(W) + \partial_y G_2(W) \leqslant 0$$

in the sense of distributions.

4. High-order finite volume schemes

4.1. Roe method

To discretize (10) the computational domain D is decomposed into subsets with a simple geometry, called cells or finite volumes: $V_i\subset\mathbb{R}^2$. It is assumed that the cells are closed convex polygons whose intersections are either empty, a complete edge or a vertex. Denote by $\mathcal{T}$ the mesh, i.e., the set of cells, and by NV the number of cells. Here, we consider rectangular structured meshes or triangular non-structured ones.

Given a finite volume V_i, $|V_i|$ will represent its area; $N_i\in\mathbb{R}^2$ its center; $\mathcal{N}_i$ the set of indexes j such that V_j is a neighbor of V_i; E_{ij} the common edge of two neighboring cells V_i and V_j, and $|E_{ij}|$ its length; d_{ij} the distance from N_i to E_{ij}; $\boldsymbol{\eta}_{ij}=(\eta_{ij,1},\eta_{ij,2})$ the normal unit vector at the edge E_{ij} pointing towards the cell V_j (see Fig. 1); Δ the maximum of the diameters of the cells; W_i^n the constant approximation to the average of the solution in the cell V_i at time t^n provided by the numerical scheme:

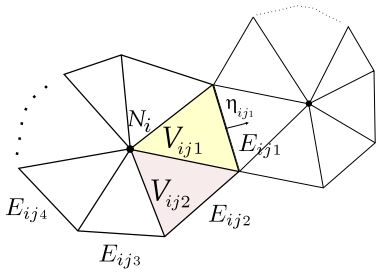


Fig. 1. Finite volume discretization.

$$W_i^n \cong \frac{1}{|V_i|}\int_{V_i} W(\mathbf{x}, t^n)\,\mathrm{d}\mathbf{x}$$

Given a family of paths Ψ, a *Roe linearization* of system (10) is a function

$$\mathcal{A}_\Psi : \Omega \times \Omega \times S^1 \to \mathcal{M}_N(\mathbb{R})$$

satisfying the following properties for each $W_L, W_R \in \Omega$ and $\boldsymbol{\eta} \in S^1$:

(i) $\mathcal{A}_\Psi(W_L, W_R, \boldsymbol{\eta})$ has N distinct real eigenvalues

$$\lambda_1(W_L, W_R, \boldsymbol{\eta}) < \lambda_2(W_L, W_R, \boldsymbol{\eta}) < \cdots < \lambda_N(W_L, W_R, \boldsymbol{\eta})$$

(ii) $\mathcal{A}_\Psi(W, W, \boldsymbol{\eta}) = \mathcal{A}(W, \boldsymbol{\eta})$.

(iii)

$$\mathcal{A}_\Psi(W_L, W_R, \boldsymbol{\eta}) \cdot (W_R - W_L) = \int_0^1 \mathcal{A}\big(\Psi(s; W_L, W_R, \boldsymbol{\eta}), \boldsymbol{\eta}\big)\frac{\partial \Psi}{\partial s}(s; W_L, W_R, \boldsymbol{\eta})\,\mathrm{d}s \tag{13}$$

We denote by $\Lambda_\Psi(W_L, W_R, \boldsymbol{\eta})$ the diagonal matrix whose coefficients are the eigenvalues $\lambda_j(W_L, W_R, \boldsymbol{\eta})$ and let $\mathcal{K}_\Psi(W_L, W_R, \boldsymbol{\eta})$ be the associated eigenvectors matrix. Let us define the positive and negative parts of $\mathcal{A}_\Psi(W_L, W_R, \boldsymbol{\eta})$ as

$$\mathcal{A}^\pm_\Psi(W_L, W_R, \boldsymbol{\eta}) = \mathcal{K}_\Psi(W_L, W_R, \boldsymbol{\eta}) \cdot \Lambda^\pm_\Psi(W_L, W_R, \boldsymbol{\eta}) \cdot \mathcal{K}_\Psi(W_L, W_R, \boldsymbol{\eta})^{-1}$$

where $\Lambda^+_\Psi(W_L, W_R, \boldsymbol{\eta})$ (respectively, $\Lambda^-_\Psi(W_L, W_R, \boldsymbol{\eta})$) is the diagonal matrix whose coefficients are the positive (respectively, negative) parts of the eigenvalues $\lambda_j(W_L, W_R, \boldsymbol{\eta})$.

In the particular case in which $\mathcal{A}_k(W)$, $k = 1, 2$, is the Jacobian matrix of a smooth flux function $F_k(W)$, property (13) does not depend on the family of paths and reduces to the usual Roe property:

$$\mathcal{A}_\Psi(W_L, W_R, \boldsymbol{\eta}) \cdot (W_R - W_L) = F_{\boldsymbol{\eta}}(W_R) - F_{\boldsymbol{\eta}}(W_L) \tag{14}$$

for any $\boldsymbol{\eta} \in S^1$.

The general expression of a Roe scheme in upwind form for solving (10) is given by [13]:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|}\sum_{j \in \mathcal{N}_i} |E_{ij}| \mathcal{A}^-_{ij} \cdot \big(W_j^n - W_i^n\big) \tag{15}$$

where

$$\mathcal{A}^-_{ij} = \mathcal{A}^-_\Psi\big(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}\big)$$

Additionally, a CFL condition must be imposed to ensure stability:

$$\Delta t \cdot \max\left\{\frac{|\lambda_{ij,k}|}{d_{ij}};\ i = 1, \ldots, NV,\ j \in \mathcal{N}_i,\ k = 1, \ldots, N\right\} = \delta \tag{16}$$

with $0 < \delta \leqslant 1$.

Remark 1. Note that in order to define the Roe solver, the spectral decomposition of the matrices $\mathcal{A}_{ij}$ must be performed. In the case of the one-layer shallow water system, the eigenvalues and eigenvectors are explicitly known, but this is not the case for the two-layer shallow water system, where a numerical algorithm must be used to compute the spectral decomposition. This fact makes the Roe solver more computationally expensive when it is applied to the two-layer shallow water system.

As in the case of systems of conservation laws, when sonic rarefaction waves appear it is necessary to modify the numerical scheme in order to obtain entropy-satisfying solutions. For instance, the Harten–Hyman entropy fix technique [16] can be easily adapted here.

Some general results concerning the consistency and well-balanced properties of Roe schemes have been studied in [13]. In [17] and [18] it has been proved that, in general, the numerical solutions provided by a path-conservative numerical scheme converge to functions which solve a perturbed system in which an error source term appears on the right-hand side. The appearance of this source term, which is a measure supported on the discontinuities, has been first observed in [19] when a scalar conservation law is discretized by means of a nonconservative numerical method. Nevertheless, in certain special situations the convergence error vanishes for finite difference methods: this is the case for systems of balance laws (see [20]). Moreover for more general problems, even when the convergence error is present, it may be only noticeable for very fine meshes, for discontinuities of large amplitude, and/or for large-time simulations: see [17,18] for details.

4.2. High-order extension

In this section we describe a high-order extension of scheme (15). Let us consider first a *reconstruction operator*, i.e., an operator that associates to a given family $\{W_i\}_{i=1}^{NV}$ of cell values two families of functions defined at the edges:

$$\gamma \in E_{ij} \mapsto W_{ij}^{\pm}(\gamma)$$

in such a way that, whenever

$$W_i = \frac{1}{|V_i|}\int_{V_i} W(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{17}$$

for some smooth function W, then

$$W_{ij}^{\pm}(\gamma) = W(\gamma) + \mathcal{O}\big(\Delta^p\big), \quad \gamma \in E_{ij}$$

It will be assumed that the reconstructions are calculated in the following way: given the family $\{W_i\}_{i=1}^{NV}$ of cell values, an approximation function is constructed at every cell V_i, based on the values W_j at some stencil of neighboring cells to V_i:

$$P_i(\mathbf{x}) = P_i\big(\mathbf{x}; \{W_j\}_{j\in\mathcal{B}_i}\big)$$

for some set of indexes $\mathcal{B}_i$. These approximation functions are usually constructed by means of interpolation or approximation methods. The reconstructions at $\gamma \in E_{ij}$ are defined as

$$W_{ij}^{-}(\gamma) = \lim_{\mathbf{x}\to\gamma} P_i(\mathbf{x}), \qquad W_{ij}^{+}(\gamma) = \lim_{\mathbf{x}\to\gamma} P_j(\mathbf{x}) \tag{18}$$

Clearly, for any $\gamma \in E_{ij}$ the following equalities are satisfied:

$$W_{ij}^{-}(\gamma) = W_{ji}^{+}(\gamma), \qquad W_{ij}^{+}(\gamma) = W_{ji}^{-}(\gamma)$$

The reconstruction operator is assumed to satisfy the following properties:

(H1) It is conservative, i.e., the following equality holds for any cell V_i:

$$W_i = \frac{1}{|V_i|}\int_{V_i} P_i(\mathbf{x})\,\mathrm{d}\mathbf{x} \tag{19}$$

(H2) It is of order p, in the sense that

$$W(\gamma) - W_{ij}^{\pm}(\gamma) = \Delta^p g_{ij}(\gamma) + \mathcal{O}\big(\Delta^{p+1}\big), \quad \gamma \in E_{ij}$$

being g_{ij} a regular function.

(H3) It is of order q in the interior of the cells, i.e., if the operator is applied to a sequence $\{W_i\}$ satisfying (17) for some smooth function $W(\mathbf{x})$, then

$$P_i(\mathbf{x}) = W(\mathbf{x}) + \mathcal{O}\big(\Delta^q\big), \quad \mathbf{x} \in \operatorname{int}(V_i) \tag{20}$$

(H4) The gradient of P_i provides an approximation of order m to the gradient of W:

$$\nabla P_i(\mathbf{x}) = \nabla W(\mathbf{x}) + \mathcal{O}\big(\Delta^m\big), \quad \mathbf{x} \in \operatorname{int}(V_i) \tag{21}$$

The semidiscrete expression of the high-order extension of scheme (15), based on a given reconstruction operator, is the following (see [13] for more details):

$$\begin{aligned} W_i'(t) = -\frac{1}{|V_i|}\Bigg[& \sum_{j\in\mathcal{N}_i} \int_{E_{ij}} \mathcal{A}_{ij}^-(\gamma,t)\big(W_{ij}^+(\gamma,t) - W_{ij}^-(\gamma,t)\big)\,\mathrm{d}\gamma \\ & + \int_{V_i} \left(\mathcal{A}_1\big(P_i^t(\mathbf{x})\big)\frac{\partial P_i^t}{\partial x}(\mathbf{x}) + \mathcal{A}_2\big(P_i^t(\mathbf{x})\big)\frac{\partial P_i^t}{\partial y}(\mathbf{x})\right)\mathrm{d}\mathbf{x}\Bigg] \end{aligned} \tag{22}$$

where P_i^t is the approximation function at time t:

$$P_i^t(\mathbf{x}) = P_i\big(\mathbf{x};\, \{W_j(t)\}_{j\in\mathcal{B}_i}\big)$$

$W_{ij}^{\pm}(\gamma,t)$ are given by

$$W_{ij}^-(\gamma,t) = \lim_{\mathbf{x}\to\gamma} P_i^t(\mathbf{x}), \qquad W_{ij}^+(\gamma,t) = \lim_{\mathbf{x}\to\gamma} P_j^t(\mathbf{x}) \tag{23}$$

and

$$\mathcal{A}_{ij}(\gamma,t) = \mathcal{A}_\Psi\big(W_{ij}^-(\gamma,t), W_{ij}^+(\gamma,t), \boldsymbol{\eta}_{ij}\big)$$

The following result can be proved (see [13]):

Theorem 4.1. *Assume that $\mathcal{A}_1$ and $\mathcal{A}_2$ are of class $\mathcal{C}^2$ with bounded derivatives and $\mathcal{A}_\Psi$ is bounded. Suppose also that the reconstruction operator satisfies hypotheses* (H1)–(H4)*. Then* (22) *is an approximation of order at least $\alpha = \min(p,q,m)$.*

Remark 2. The conclusion of Theorem 4.1 is rather pessimistic: the observed order in experiments is usually $\alpha = \min(p,q,m+1)$. See [13] for more details.

In practice, the integral terms in (22) must be approximated numerically. A one-dimensional quadrature formula of order $\bar{r}$ is applied to calculate the line integrals:

$$\int_a^b f(s)\,\mathrm{d}s = (b-a)\left(\sum_{l=1}^{n(\bar{r})} \omega_l f(x_l)\right) + \mathcal{O}\big(\Delta^{\bar{r}}\big) \tag{24}$$

while a two-dimensional quadrature formula of order $\bar{s}$ is used to compute the volume integrals:

$$\int_{V_i} f(\mathbf{x})\,\mathrm{d}\mathbf{x} = |V_i| \sum_{l=1}^{n(\bar{s})} \alpha_l f\big(\mathbf{x}_l^i\big) + \mathcal{O}\big(|V_i|^{\bar{s}}\big) \tag{25}$$

To preserve the order of the numerical scheme, it is necessary to have $\bar{r} \geqslant \alpha$ and $\bar{s} \geqslant \alpha$.

Finally, the numerical scheme is written as follows:

$$\begin{aligned} W_i'(t) = -\frac{1}{|V_i|}\Bigg[& \sum_{j\in\mathcal{N}_i} |E_{ij}| \sum_{l=1}^{n(\bar{r})} w_l \mathcal{A}_{ij,l}^-(t)\big(W_{ij,l}^+(t) - W_{ij,l}^-(t)\big) \\ & + |V_i| \sum_{l=1}^{n(\bar{s})} \alpha_l \left(\mathcal{A}_1\big(P_i^t\big(\mathbf{x}_l^i\big)\big)\frac{\partial P_i^t}{\partial x}\big(\mathbf{x}_l^i\big) + \mathcal{A}_2\big(P_i^t\big(\mathbf{x}_l^i\big)\big)\frac{\partial P_i^t}{\partial y}\big(\mathbf{x}_l^i\big)\right)\Bigg] \end{aligned} \tag{26}$$

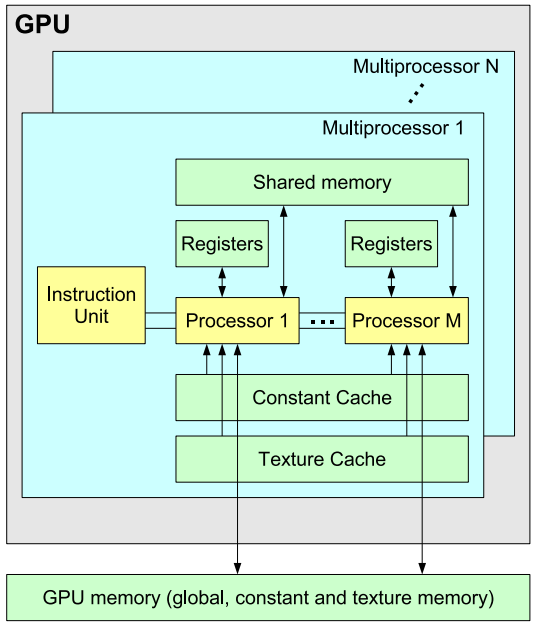


Fig. 2. GPU architecture.

where

$$W^{\pm}_{ij,l}(t) = W^{\pm}_{ij}\left(a_{ij} + s_l(b_{ij} - a_{ij}), t\right)$$

and

$$\mathcal{A}_{ij,l}(t) = \mathcal{A}_{\Psi}\left(W^{-}_{ij,l}(t), W^{+}_{ij,l}(t), \boldsymbol{\eta}_{ij}\right)$$

being a_{ij} and b_{ij} the vertices of edge E_{ij}.

Remark 3. A technique that avoids the explicit computation of $\nabla P_i(\mathbf{x})$ has been introduced in [21] in the one-dimensional case. The use of this technique, that is based on the trapezoidal rule and Romberg extrapolation, makes the expected order of accuracy to be $\min(p, q)$. The extension to two-dimensional problems is straightforward for structured meshes, while for unstructured meshes a Romberg extrapolation formula for triangles can be used (see [22]).

For time stepping, high-order TVD Runge–Kutta methods like those described in [23] are applied. In particular, in this work we use a third-order reconstruction operator in space and a third-order TVD Runge–Kutta method to advance in time.

The reconstruction operator used here is the one proposed in [24]: it is a compact reconstruction operator of polynomial type, that is third-order accurate on each computational cell and it can be defined in general non-uniform quadrilateral meshes.

The well-balancedness properties of schemes (22) and (26) have been analyzed in [13].

5. CUDA architecture and programming model

According to the CUDA framework, both the CPU and the GPU maintain their own memory. It is possible to copy data from CPU memory to GPU memory and vice versa.

The GPU is formed by a set of Single Instruction Multiple Data (SIMD) multiprocessors, each one having 32 processors ($M = 32$) in Fermi architecture and 8 processors ($M = 8$) in GT200 architecture (see Fig. 2). At any clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

A function executed on the GPU is called a *kernel*. A kernel is executed by many threads which are organized forming a grid of thread blocks that run logically in parallel [25]. All blocks and threads have spatial indices, so that the spatial position of each thread could be identified in the program. Each thread block runs in a single multiprocessor. A *warp* is the number of threads that can run concurrently in a multiprocessor. Warp size is 32 threads. Each block is split into warps, and periodically a scheduler switches from one warp to another. This allows to hide the high latency when accessing the GPU memory, since some threads can continue their execution while other threads are waiting.

A thread that executes on the GPU has access to the following memory spaces:

- *Registers*: Each thread has its own readable and writable registers.
- *Shared memory*: Shared by all threads of a block. Readable and writable only from the GPU. In Fermi architecture, its size is configurable and can be 16 or 48 KB. It is faster than global memory.
- *Global memory*: Shared by all blocks of a grid. Readable and writable from CPU and GPU. It is slow due to its high latency.
- *Constant memory*: Shared by all blocks of a grid. Readable from GPU and writable from CPU. Its size is 64 KB and it is cached, making it faster than global memory if the data is in cache. Cache size is 8 KB per multiprocessor.
- *Texture memory*: Shared by all blocks of a grid. Readable from GPU and writable from CPU. It is cached and optimized from 2D spatial locality, i.e. it is especially suited for each thread to access its closer neighborhood in texture memory. Cache size varies between 6 and 8 KB per multiprocessor.

In Fermi architecture, each multiprocessor has 32 768 registers, which are split and assigned to the threads that execute concurrently on that multiprocessor.

6. Structured meshes

In this section we outline the parallelization of the numerical scheme and describe the details of our CUDA implementations for dealing with the high-order numerical scheme for the one-layer shallow water system on structured meshes.

6.1. Parallelization

In this section we briefly describe the potential data parallelism of the numerical scheme described in the previous section and its implementation in CUDA.

Initially, the finite volume mesh must be constructed from the input data with the appropriate setting of initial and boundary conditions. Then the time stepping is performed by applying a third-order Runge–Kutta TVD method, consisting on three steps. At each step, the spatial discretization (26) must be performed as follows:

(i) **Reconstruction and volume integral computation:** First a reconstruction procedure at each cell and for each variable must be performed to define the functions $P_i(\mathbf{x})$. Next, the numerical approximation of the volume integral is computed using a third-order Gaussian quadrature formula

$$\Sigma_i = -|V_i| \sum_{l=1}^{n(\bar{s})} \alpha_l \left(\mathcal{A}_1\big(P_i^t(\mathbf{x}_l^i)\big) \frac{\partial P_i^t}{\partial x}(\mathbf{x}_l^i) + \mathcal{A}_2\big(P_i^t(\mathbf{x}_l^i)\big) \frac{\partial P_i^t}{\partial y}(\mathbf{x}_l^i) \right)$$

The reconstructed values $W_{ij,l}$ at the quadrature points of each edge of the cell V_i are also computed. Again, a third-order Gaussian quadrature formula is used. Therefore, two values must be computed at each edge of V_i.

(ii) **Edge-based calculations:** The following computations must be performed at each edge E_{ij} common to cells V_i and V_j, using the reconstructed values $W_{ij,l}^-$ and $W_{ij,l}^+$ previously computed:

$$\Sigma_{ij}^{\pm} = |E_{ij}| \sum_{l=1}^{n(\bar{r})} w_l \mathcal{A}_{ij,l}^-(t)\big(W_{ij,l}^+(t) - W_{ij,l}^-(t)\big)$$

(iii) **Volume-based calculations:** At each cell V_i, the following computations must be performed:
  (a) Computation of the local Δt_i for each volume.
  (b) Computation of $W_i^{n+1,s}$: The $n+1$, s-th state of each volume must be approximated from the n-th and the $n+1$, $(s-1)$-th states using the data computed at the previous steps.

Several remarks can be made related to the description of the parallel algorithm. The computation steps required by the problem addressed here can be classified into two groups: computations associated to edges and computations associated to volumes. The scheme exhibits a high degree of data parallelism because the computation at each edge/volume is independent with respect to the computation performed at the rest of edges/volumes. Moreover, the scheme presents a high arithmetic intensity and the computation exhibits a high degree of locality. These remarks indicate that this problem is suitable for being implemented on GPUs using CUDA.

Concerning the implementation, each processing step previously described is assigned to a CUDA kernel. Let us describe the implementation of a high-order scheme for the one-layer shallow water system on structured meshes using CUDA. More details can be found in [9] and [24].

- **Build the data structure:** For each volume, we store its state $W = [U, H]$. In the case of the one-layer shallow water system $U = [h, q_x, q_y]$. We define an array of `float4`[1] elements, where each element represents a volume and contains

[1] The `float4` data type represent structures with four single precision real components.

the former parameters. This array is stored as a 2D texture since texture memory is especially suited for each thread to access its closer environment in texture memory. The per-block shared memory, on the other hand, is more suitable when each thread needs to access many elements located in global memory, and each thread of a block loads a small part of these elements into shared memory. We first implemented a CUDA program using shared memory instead of a texture, where each thread of a block loaded the data of a volume into shared memory, but later we got better execution times by using a texture.

The area of the volumes and the length of the vertical and horizontal edges are precalculated and passed to the CUDA kernels that need them.

We can know at runtime if an edge or volume is frontier or not and the value of $\boldsymbol{\eta}_{ij}$ at an edge by checking the position of the thread in the grid.

- **Reconstruction and integral computation:** In this step, the reconstruction values $U^{\pm}_{ij,l}$, $l = 1, 2$, as well as the reconstructed topography $H^{\pm}_{il,j}$ are computed and stored in four arrays located in global memory, each one being an array of `float4` elements. The size of each array is twice the number of volumes and they are associated to the four edges of a cell (south, north, east and west). Moreover, the integral term Σ_i is also computed and stored in an accumulator placed in global memory. This accumulator is an array of `float4` elements and its size is the number of volumes. This accumulator is also used to store the contributions of the vertical edges. In this process, each thread represents a finite volume cell.
- **Process vertical and horizontal edges:** We divide the edge processing into vertical and horizontal edge processing. For vertical edges $\eta_{ij,y} = 0$, and for horizontal edges $\eta_{ij,x} = 0$. Therefore, all the operations where these terms take part can be avoided, thus increasing efficiency.

  Here, each thread represents a vertical or a horizontal edge, and computes the contribution to their adjacent volumes. The edges (i.e., threads) synchronize each other when contributing to a particular volume by means of two accumulators stored in global memory, each one being an array of `float4` elements. Note that one of them has been previously used to store the integral cell computation. The size of each accumulator is the number of volumes. Each element of the accumulators stores the edge contributions to the volume (a 3×1 vector, $\Sigma^{\pm}_{ij}$, and a `float` value storing $\|\Lambda_{ij}\|_{\infty}$). In the processing of vertical edges, each edge writes the contribution to its right-side volume in the first accumulator, and the contribution to its left-side volume in the second accumulator. Next, the processing of horizontal edges is performed in an analogous way, with the difference that the contribution is added to the accumulators.
- **Compute Δt_i for each volume:** In this step, each thread represents a volume and the local Δt_i of the volume V_i is computed using the CFL condition (16).
- **Get the minimum Δt:** This step finds the minimum of the local Δt_i of the volumes by applying a reduction algorithm on the GPU. The reduction algorithm applied is the kernel 7 (the most optimized one) of the reduction sample included in the CUDA Software Development Kit [8].
- **Compute $U_i^{n+1,s}$ for each volume:** In this step, each thread represents a volume and the state U_i of the volume V_i is updated. The final value is obtained by adding up the two 3×1 vectors stored in the positions corresponding to the volume V_i in both accumulators (Note that the topography is an artificial unknown and it does not change during the computation). Since a CUDA kernel cannot directly write into textures, the texture is initially updated by writing the results into a temporal array, which is then copied to the CUDA array bound to the texture.

6.2. Numerical experiments

Different implementations of the scheme have been performed: a sequential CPU code was written in C++ using double precision using the Eigen library [26], denoted by 'CPU 1 core'. A quadcore CPU code using OpenMP [27] denoted by 'CPU 4 cores', a GPU code implemented using Cg [6] and single precision, denoted by 'Cg', a GPU code implemented in CUDA using single precision, denoted by 'CUSP', and a GPU code implemented in CUDA using double precision, denoted by 'CUDP'. The CPU was an Intel Xeon E5430 (2.66 GHz 12 MB L2 Cache), while two different GPUs have been used: a NVIDIA GeForce GTX 260 (216 stream processors with 896 MB) and a NVIDIA GeForce GTX 280 (240 stream processors with 1 GB).

As test problem, we consider a circular dambreak problem in the $[-5, 5] \times [-5, 5]$ domain. The depth function is $H(x, y) = 1 - 0.4e^{-x^2-y^2}$ and the initial condition is:

$$W_i(\boldsymbol{x}, 0) = \begin{pmatrix} h(\boldsymbol{x}, 0) \\ 0 \\ 0 \end{pmatrix}, \quad \text{where } h(\boldsymbol{x}, 0) = \begin{cases} 1 + H(\boldsymbol{x}) & \text{if } \sqrt{x^2 + y^2} > 0.6 \\ 3 + H(\boldsymbol{x}) & \text{otherwise} \end{cases}$$

The numerical schemes are run for different mesh sizes. Simulations are carried out in the time interval [0,1]. CFL parameter is $\delta = 0.9$ and wall boundary conditions ($\boldsymbol{q} \cdot \boldsymbol{\eta} = 0$) are considered. Fig. 3 shows the evolution of the free surface ($\eta = h - H$) computed for the second mesh using the high order scheme.

Tables 1 and 2 show the execution times in seconds for all the meshes and programs. As can be seen, the execution times seems to grow linearly with the number of volumes of the mesh, as expected. Figs. 4 and 5 show graphically the speedup obtained in all the implementations with respect to the monocore version.

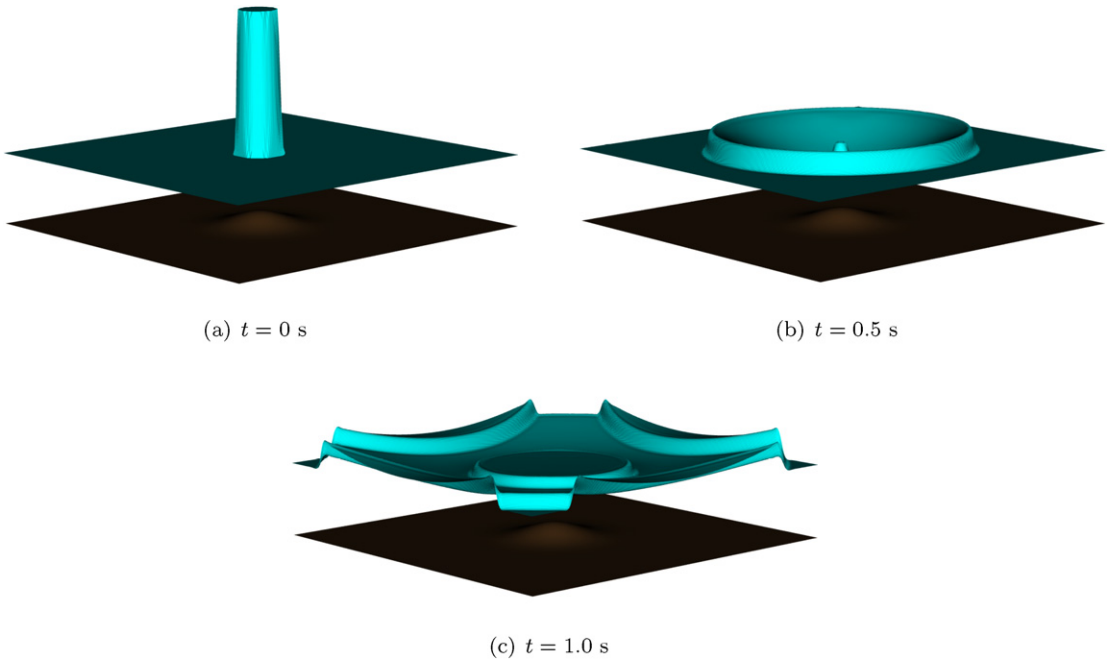


Fig. 3. Evolution of the free surface: third order numerical scheme.

Table 1
Structured meshes: Execution times in seconds for all meshes and programs (first order).

Volumes	CPU		GTX 260			GTX 280		
	1 core	4 cores	Cg	CUSP	CUDP	Cg	CUSP	CUDP
100×100	0.8	0.26	0.11	0.01	0.06	0.08	0.01	0.05
200×200	6.7	1.98	0.26	0.06	0.37	0.2	0.06	0.32
400×400	56.6	26.57	0.84	0.39	2.75	0.68	0.35	2.34
800×800	455.9	216.5	4.42	2.91	21.43	3.75	2.48	18.49
1600×1600	3639.9	1722.8	30.72	23.44	167.2	26.14	19.27	143.0
2000×2000	7135.7	3375.4	58.54	44.87	336.9	49.48	38.34	272.0

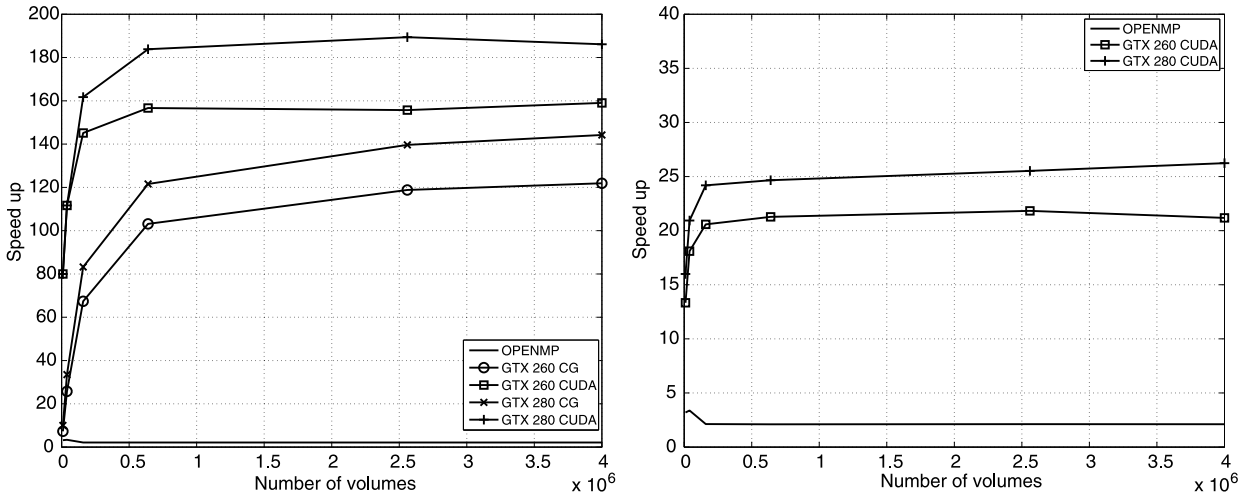
Table 2
Structured meshes: Execution times in seconds for all the meshes and programs (third order).

Volumes	CPU	GTX 260	GTX 280
	1 core	CUSP	CUSP
100×100	2.36	0.025	0.024
200×200	19.0	0.18	0.15
400×400	152.10	1.36	1.31
800×800	1218.32	10.89	9.47
1200×1200	4068.0	36.16	30.01

Concerning the first order numerical scheme on structured meshes (see Table 1 and Fig. 4) we can see that the execution times of the single precision CUDA program (CUSP) outperform that of Cg in all cases with both graphics cards. Using a GTX 280, for big problems, CUSP achieves a speedup of two orders of magnitude with respect to the monocore version, reaching a performance gain of more than 180 (see Fig. 4(a)). The double precision CUDA program (CUDP) has been about 7 times slower than CUSP for big problems in both graphics cards (see Fig. 4(b)), which seems logical considering that, in GT200 architecture, each multiprocessor has 8 single precision units and only one double precision unit. As expected, the OpenMP version only reaches a speedup less than four with respect to the monocore program in all meshes (see Fig. 4).

Similar results are obtained for the high-order numerical scheme on structured meshes (see Table 2 and Fig. 5), reaching a performance of about 140 with respect to the monocore version.

We also have compared the numerical solutions obtained in the monocore and the CUDA programs. The L^1 norm of the difference between the solutions obtained in CPU and GPU at time $t = 1.0$ for all meshes was calculated. The order of magnitude of the L^1 norm using CUSP varies between 10^{-4} and 10^{-6}, while that obtained using CUDP varies between 10^{-12} and 10^{-14}, which reflects the different accuracy of the numerical solutions computed on the GPU using single and double precision.



(a) Single precision (b) Double precision

Fig. 4. First order scheme: speedup on structured meshes. Single precision (left). Double precision (right).

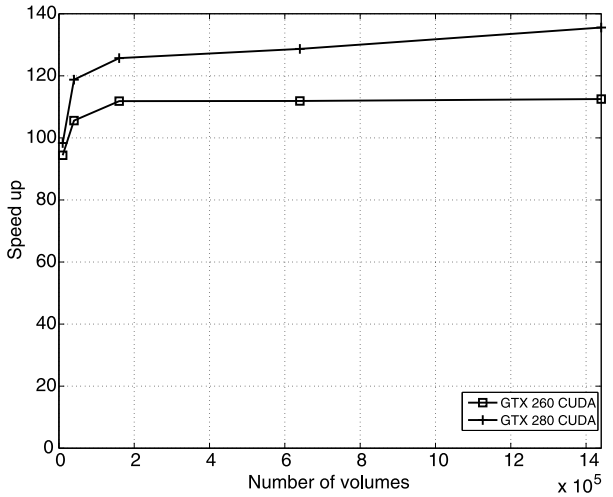


Fig. 5. High-order scheme: speedup on non-structured meshes (single precision).

7. Unstructured triangular meshes

In this section we outline the parallelization of the numerical scheme and describe the details of our CUDA implementations for dealing with one and two-layer first order shallow water systems on triangular meshes.

7.1. Parallelization

Fig. 6 shows a graphical description of the parallel algorithm, obtained from the mathematical description of the numerical scheme. In this figure, the main calculation phases are identified with circled numbers and the main sources of data parallelism are represented with overlapping rectangles indicating that the calculation affected by it can be performed simultaneously for each data item of a set (the data items can represent the volumes or the edges of the finite volume mesh). The arrows connecting two computing phases represent data dependencies between the two phases.

Initially, the finite volume mesh is constructed from the input data. Then the time stepping process is repeated until the final simulation time is reached. At the $(n+1)$-th time step, Eq. (15) must be evaluated to update the state of each cell. In order to add the contributions associated to each edge, two variables are used in the algorithm for each volume V_i $(i = 1, \ldots, L)$: Z_i is used to store the contributions to the local time step size of the volume V_i and M_i is used to store the sum of the contributions to the state of V_i. M_i is a 3×1 vector for the one-layer case and a 6×1 vector for the two-layer case. The type of the rest of variables which appear in Fig. 6 also depends on the case.

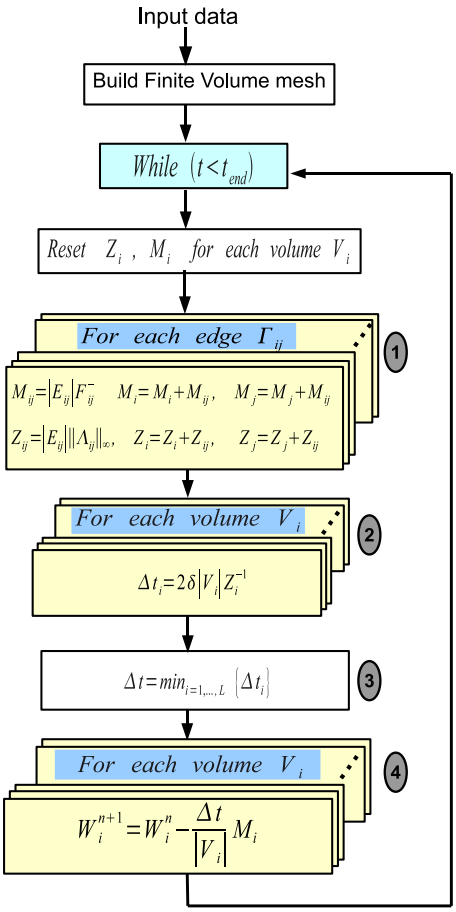


Fig. 6. Main calculation phases in the parallel algorithm.

Each of the main calculation phases of the evaluation present a high degree of parallelism because the computation at each edge or volume is independent with respect to that performed or associated to the other edges or volumes:

1. **Edge-based calculations:** This is the most costly phase in the algorithm and involves two calculations for each edge E_{ij} communicating two cells V_i and V_j ($i, j \in \{1, \dots, L\}$):
   (a) Vector $M_{ij} = |E_{ij}|F_{ij}^-$, where $F_{ij}^- = \mathcal{A}_{ij}^- \cdot (W_j^n - W_i^n)$ in Eq. (15), must be computed as the contribution of each edge to the sum associated to the adjacent cells V_i and V_j. This contribution can be computed independently for each edge and must be added to the partial sums associated to each cell (M_i and M_j).
   In the one-layer-case, to compute F_{ij}^-, the mathematical expression for the matrices Λ_{ij} and $\mathcal{K}_{ij}$ is known, and these matrices are computed directly by evaluating these expressions. On the other hand, in the two-layer case, the matrices are obtained by using a particular algorithm to find the eigenvalues and eigenvectors of the matrix $\mathcal{A}_{ij} \in \mathbb{R}^{6\times 6}$.
   (b) The value $Z_{ij} = |E_{ij}|\|\Lambda_{ij}\|_\infty$ can be computed independently for each edge and added to the partial sums associated to each cell (Z_i and Z_j) as an intermediate step to compute the n-th time step Δt^n.
2. **Computation of the local Δt for each volume:** In practice, Eq. (16) can be replaced by the condition $\Delta t = \min\{\Delta t_i\}$, being $\Delta t_i = 2\delta|V_i|Z_i^{-1}$. For each volume V_i, the local Δt is computed. The computation for each volume does not depend on the computation for the rest of volumes and therefore this phase can be performed in parallel.
3. **Computation of Δt^n:** The minimum of all the local Δt values previously obtained for each volume must be computed.
4. **Computation of W_i^{n+1}:** The $(n+1)$-th state of each volume (W_i^{n+1}) must be approximated from the n-th state using the data computed in the previous phases. This phase can also be completed in parallel (see Fig. 6).

Since the numerical scheme exhibits a high degree of potential data parallelism (mainly in phases 1, 2 and 4), it is a good candidate to be implemented on CUDA architectures.

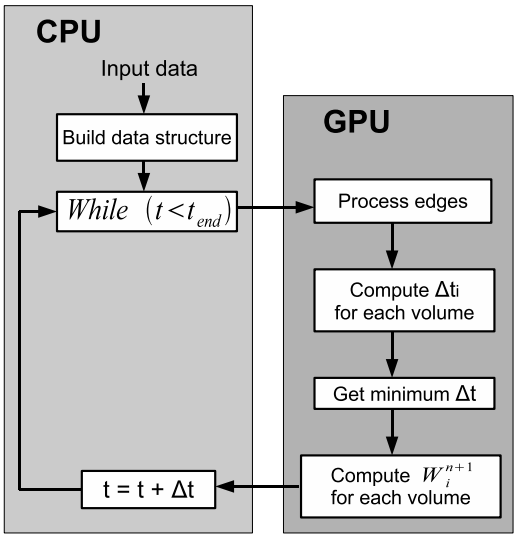


Fig. 7. General steps of the CUDA algorithm for one and two-layer.

7.2. One-layer

7.2.1. Details of the CUDA implementation

In this section we describe the CUDA implementation of the parallel algorithm we have developed for one-layer shallow water system using single precision that we denoted CUSP hereafter. It is a variant of the algorithm described in [9] for triangular meshes. The general steps of the parallel algorithm are depicted in Fig. 7. Each processing step executed on the GPU is assigned to a CUDA kernel. Next, we describe in detail each step:

- **Build data structure:** In this step, the data structure that will be used on the GPU is built. For each volume, we store its initial state (h, q_x and q_y), its depth H and its area. We define an array of `float4` elements and another array of `float` elements. The size of both arrays is the number of volumes. The first array contains h, q_x, q_y and H, and is stored as 1D texture. The second array contains the area of the volumes and is stored in global memory.
  For each edge, we store its normal $(\eta_{ij,x}, \eta_{ij,y})$, the positions of the volumes V_i and V_j in the former arrays, and the two accumulators (denoted by an `int` value: 1, 2 or 3) where the edge must write its contributions to the volumes V_i and V_j, respectively. We use two arrays in global memory, where the size of each array is the number of edges: an array of `float2` elements to store the normal, and another array of `int4` elements to store the last four integer values.
  We can know at runtime if an edge is frontier by checking if the value of the position of the volume V_j is -1.
- **Process edges:** In [9], since we were working with regular meshes, we divided the edge processing into vertical and horizontal edge processing, hence allowing some terms of the numerical scheme to be removed, increasing efficiency. For triangular meshes, this is not possible. Therefore, we must compute the whole numerical scheme and we use a single kernel for processing all the edges.
  In the edge processing, each thread represents an edge, and computes the contribution of the edge to their adjacent volumes as described in Section 7.1.
  The edges (i.e. threads) synchronize each other when contributing to a particular volume by means of three accumulators (in [9] we used two accumulators for regular meshes), each one being an array of `float4` elements. The size of each accumulator is the number of volumes. Each element of the accumulators stores the contributions of the edges to W_i (the 3×1 vector M_i in Fig. 6) and to the local Δt of the volume (the `float` value Z_i in Fig. 6). Fig. 8 shows this process graphically.
- **Compute Δt_i for each volume:** In this step, each thread represents a volume and computes the local Δt_i of the volume V_i as described in Section 7.1. The final Z_i value is obtained by adding the three float values stored in the positions corresponding to the volume V_i in the accumulators.
- **Get minimum Δt:** This step finds the minimum of the local Δt_i of the volumes by applying a reduction algorithm on the GPU. The reduction algorithm applied is the kernel 7 (the most optimized one) of the reduction sample included in the CUDA Software Development Kit [8].
- **Compute W_i for each volume:** In this step, each thread represents a volume and updates the state W_i of the volume V_i as described in Section 7.1. The final M_i value is obtained by adding the three 3×1 vectors stored in the positions corresponding to the volume V_i in the accumulators. Since the 1D texture containing the volume data is stored in linear memory, we update the texture by writing directly into it.

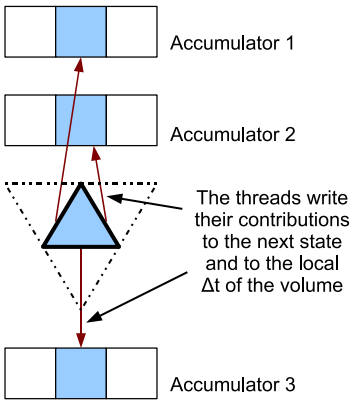


Fig. 8. Computing the sum of the contributions of the edges of each volume for the one-layer case.

As in the case of structured meshes, several implementations have been performed. Let us remark that the data structure designed for the CUDP implementation has same differences with respect to the CUSP implementation: the volume data is stored in two arrays of `double2`[2] elements (which contain the state and the depth of the volumes) and one array of `double` elements (which contain the area of the volumes). The normal vectors of the edges are stored in an array of `double2` elements. We use three accumulators of `double2` elements, where the size of each accumulator is twice the number of volumes.

7.2.2. Experimental results

We consider a test problem consisting in a circular dambreak problem in the $[-5,5]\times[-5,5]$ rectangular domain. The depth function is given by $H(\boldsymbol{x}) = 4 - 1.5e^{-x^2-y^2}$ and the initial condition is:

$$W_i(\boldsymbol{x},0) = \big(h(\boldsymbol{x},0),\ 0,\ 0\big)^{\mathrm{T}}$$

where

$$h(\boldsymbol{x},0) = \begin{cases} 2.5 & \text{if } \sqrt{x^2+y^2} > 1 \\ 5 & \text{otherwise} \end{cases}$$

The numerical scheme is run for several triangular finite volume meshes with different number of volumes (see Table 3). Simulation is carried out in the time interval $[0,1]$. CFL parameter is $\delta = 0.9$ and wall boundary conditions ($\boldsymbol{q}\cdot\boldsymbol{\eta} = 0$) are considered.

The four implementations previously described have been used to perform the numerical experiments. All the programs were executed on a Core i7 920 with 4 GB RAM. Graphics cards used were a GeForce GTX 260 and a GeForce GTX 480. The evolution of the free surface ($\eta = h - H$) computed with the third mesh is shown in Fig. 9 for several time steps.

Table 3 shows the execution times in seconds for all the meshes and programs. As can be seen, the number of volumes and the execution times scale with a different factor because the number of time steps required for the same time interval also augments when the number of cells is increased. Fig. 10 shows graphically the speedups obtained in the CUDA and OpenMP implementations with respect to the monocore CPU version using both graphics cards. For big meshes, CUSP achieves a speedup of approximately 150 in the GTX 480 card, and a speedup of 80 in the GTX 260. On the other hand, CUDP reaches a speedup of 40 in the GTX 480 card, and 13 in the GTX 260. The OpenMP version has only reached a speedup of 2.6 for big meshes. CUDP has been 3.6 times slower than CUSP for big meshes in the GTX 480 card, and 6.1 times slower in the GTX 260. As can be seen, the speedups reached with both CUDA programs are worse than those obtained in [9] applying the same numerical scheme on regular meshes.

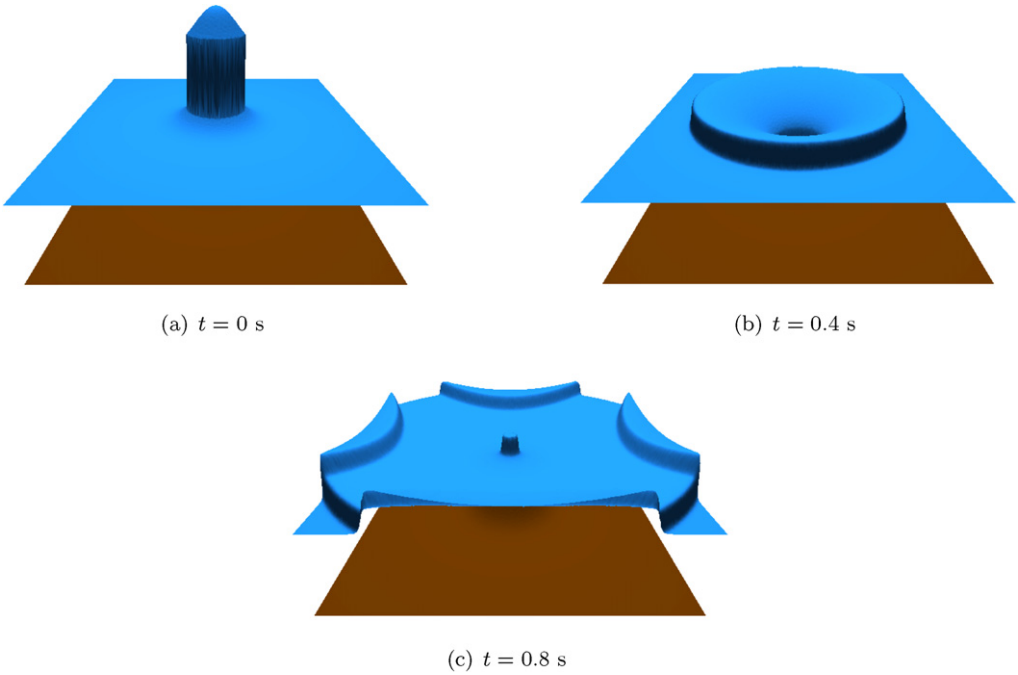
In the GTX 480 card, we get better execution times by setting the sizes of the L1 cache and shared memory to 48 KB and 16 KB per multiprocessor, respectively, for the edge processing CUDA kernel.

We also have compared the numerical solutions obtained in the monocore and the CUDA programs. The L^1 norm of the difference between the solutions obtained in CPU and GPU at time $t = 1.0$ for all meshes was calculated. The order of magnitude of the L^1 norm using CUSP varies between 10^{-3} and 10^{-5}, while that of obtained using CUDP varies between 10^{-13} and 10^{-14}, which reflects the different accuracy of the numerical solutions computed on the GPU using single and double precision.

[2] The `double2` data type represents structures with two double precision real components. Note that `float4` and `double2` data types require the same amount of memory (16 bytes).

Table 3
Execution times in seconds for all the meshes and programs for the one-layer case.

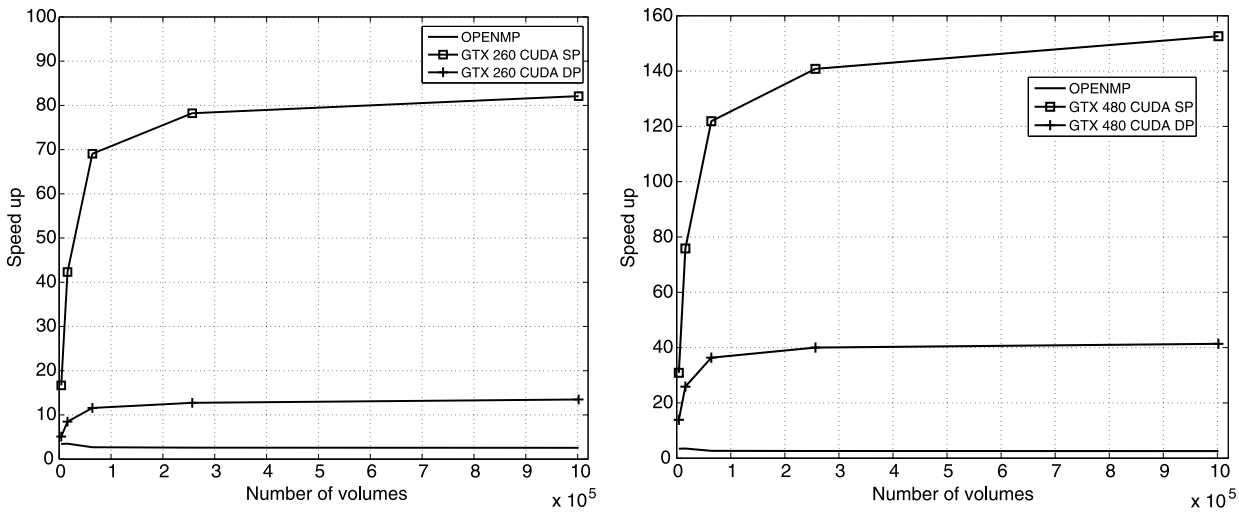
Mesh size	CPU		GTX 260		GTX 480	
	1 core	4 cores	CUSP	CUDP	CUSP	CUDP
4016	0.25	0.073	0.015	0.049	0.0081	0.018
16 040	2.20	0.63	0.052	0.26	0.029	0.085
64 052	20.72	7.75	0.30	1.79	0.17	0.57
256 576	176.0	67.87	2.25	13.83	1.25	4.40
1 001 898	1408.6	551.8	17.16	104.4	9.23	34.06



(a) $t = 0$ s (b) $t = 0.4$ s

(c) $t = 0.8$ s

Fig. 9. Evolution of the free surface.



(a) GTX 260 (b) GTX 480

Fig. 10. Speedups obtained in the CUDA and OpenMP implementations in all meshes for the one-layer case.

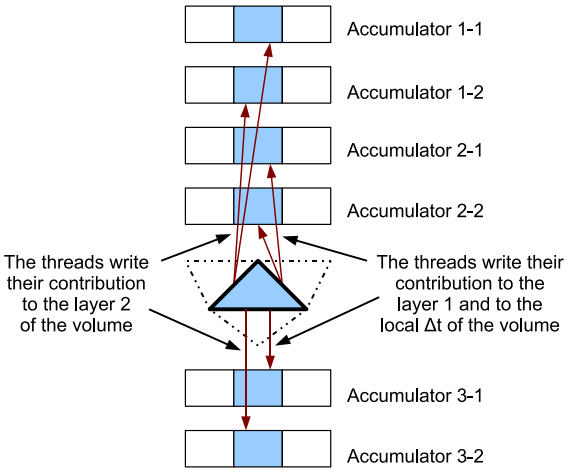


Fig. 11. Computing the sum of the contributions of the edges of each volume for the two-layer case.

7.3. Two-layer

7.3.1. Details of the CUDA implementation

In this section we describe the CUDA implementation of the parallel algorithm we have developed for two-layer shallow water system. It is a variant of the algorithm described in [10] for triangular meshes. The general steps of the parallel algorithm are the same of the one-layer case and are depicted in Fig. 7. Next, we describe the differences with respect to the one-layer implementation detailed in Section 7.2.1:

- **Build data structure:** For each volume, we store its initial state (h_1, $q_{1,x}$, $q_{1,y}$, h_2, $q_{2,x}$ and $q_{2,y}$), its depth H and its area. We define two arrays of `float4` elements, where each element represents a volume. The first array contains h_1, $q_{1,x}$, $q_{1,y}$ and H, while the second array contains h_2, $q_{2,x}$, $q_{2,y}$ and the area. Both arrays are stored as 1D textures.
  The data stored for each edge and the arrays used for this purpose are the same as the one-layer implementation.
- **Process edges:** As in the one-layer case, in this step each thread represents an edge, and computes the contribution of the edge to their adjacent volumes as described in Section 7.1.
  The edges (i.e. threads) synchronize each other when contributing to a particular volume by means of six accumulators (in [10] we used four accumulators for regular meshes), each one being an array of `float4` elements. The size of each accumulator is the number of volumes. Let us call the accumulators 1-1, 1-2, 2-1, 2-2, 3-1 and 3-2. Each element of accumulators 1-1, 2-1 and 3-1 stores the contributions of the edges to the layer 1 of W_i (the first 3 elements of M_i) and to the local Δt of the volume (a `float` value Z_i), while each element of accumulators 1-2, 2-2 and 3-2 stores the contributions of the edges to the layer 2 of W_i (the last 3 elements of M_i). Fig. 11 shows this process graphically.
- **Compute Δt_i for each volume:** In this step, each thread represents a volume and computes the local Δt_i of the volume V_i as described in Section 7.1. The final Z_i value is obtained by adding the three float values stored in the positions corresponding to the volume V_i in accumulators 1-1, 2-1 and 3-1.
- **Get minimum Δt:** Equal to the one-layer implementation.
- **Compute W_i for each volume:** In this step, each thread represents a volume and updates the state W_i of the volume V_i as described in Section 7.1. The final M_i value is obtained as follows: the first 3 elements of M_i (the contribution to layer 1) are obtained by adding the three 3×1 vectors stored in the positions corresponding to the volume V_i in accumulators 1-1, 2-1 and 3-1, while the last 3 elements of M_i (the contribution to layer 2) are obtained by adding the two 3×1 vectors stored in the positions corresponding to the volume V_i in accumulators 1-2, 2-2 and 3-2. Since the 1D textures containing the volume data are stored in linear memory, we update the textures by writing directly into them.

Again, several implementations have been performed to compare the efficiency of the CUDA programs: in this particular case the CUSP implementation does not provide good results. A mixed precision CUDA implementation (CUSDP) is considered. In this case, the eigenvalues and eigenvectors of the $\mathcal{A}_{ij}$ matrix are computed using double precision to avoid numerical instability problems, but the rest of operations are performed in single precision.

In CUDP implementation, the volume data is stored in three arrays of `double2` elements (which contain the state of the volumes), and another array of `double2` elements to store the depth H and the area. The normal vectors of the edges are stored in an array of `double2` elements. We use three accumulators of `double2` elements to store the contributions to W_i, where the size of each accumulator is three times the number of volumes. We also use three accumulators of

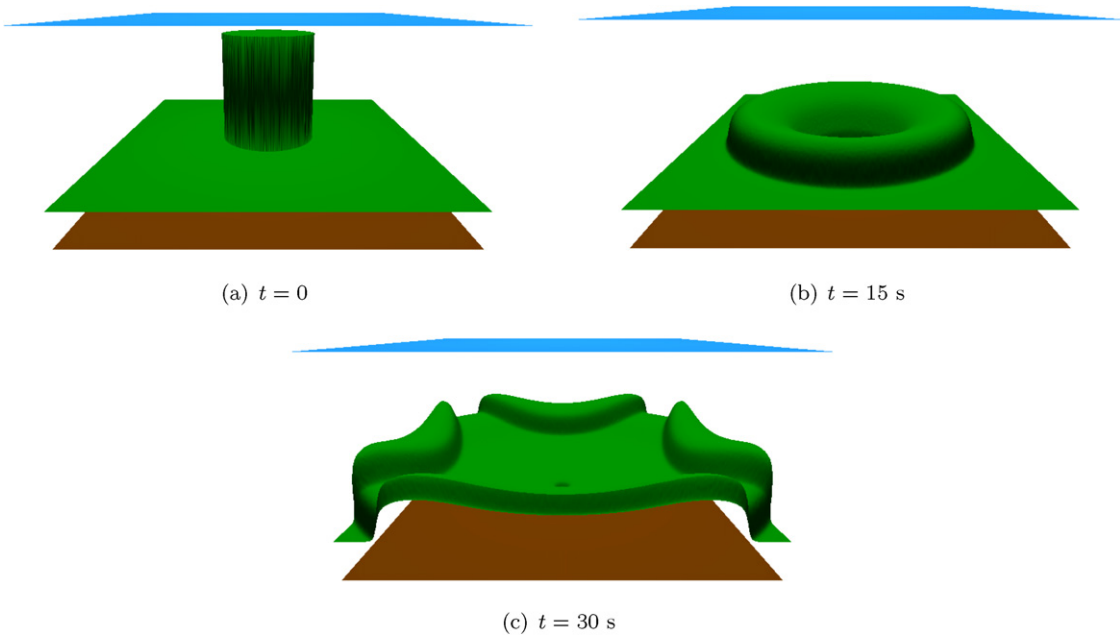


Fig. 12. Evolution of the free surface and interface (two-layer fluid).

`double` elements to store the contributions to the local Δt of each volume, where the size of each accumulator is the number of volumes.

7.3.2. Experimental results

We consider an internal circular dambreak problem in the $[-5,5]\times[-5,5]$ rectangular domain in order to compare the performance of our implementations. The depth function is given by $H(\boldsymbol{x})=5$ and the initial condition is:

$$W_i(\boldsymbol{x},0)=\big(h_1(\boldsymbol{x},0),\ 0,\ 0,\ h_2(\boldsymbol{x},0),\ 0,\ 0\big)^{\mathrm{T}}$$

where

$$h_1(\boldsymbol{x},0)=\begin{cases}4 & \text{if } \sqrt{x^2+y^2}>1.5,\\ 0.5 & \text{otherwise,}\end{cases}\qquad h_2(\boldsymbol{x},0)=5-h_1(\boldsymbol{x},0)$$

The numerical scheme is run for several triangular finite volume meshes with different number of volumes (see Table 4). Simulation is carried out in the time interval $[0,1]$. CFL parameter is $\delta=0.9$, $r=0.998$ and wall boundary conditions ($\boldsymbol{q}_1\cdot\boldsymbol{\eta}=0$, $\boldsymbol{q}_2\cdot\boldsymbol{\eta}=0$) are considered.

In the CUDA implementations, the eigenvalues and eigenvectors of the A_{ij} matrix are computed using the *rg* subroutine of the EISPACK library [28], converted to C code using the `f2c` utility.

All the programs were executed on a Core i7 920 with 4 GB RAM. Graphics card used was a GeForce GTX 260.

Fig. 12 shows the evolution of the free surface ($\eta=h_1+h_2-H$) and the interface ($\eta_i=h_2-H$) of the fluid computed using the third mesh, for several time steps.

Table 4 shows the execution times in seconds for all the meshes and programs. Fig. 13 shows graphically the speedups obtained in the CUDA and OpenMP implementations with respect to the monocore CPU version. For big meshes, CUSDP achieves a speedup of 10, while CUDP reaches a speedup of more than 6. The OpenMP version has reached a speedup of 3.2 for big meshes. It is worth noting that for small meshes, CUDP has got slightly better execution times than CUSDP, although for big meshes CUSDP has been faster. As it happened in the one-layer case, the speedups reached with both CUDA programs are worse than those obtained in [10] applying the same numerical scheme on regular meshes.

As can be seen, the speedups reached with the CUSDP implementation are notably worse than those obtained for CUSP in one-layer systems. This is mainly due to two reasons. Firstly, since double precision has been used to compute the eigenvalues and eigenvectors (see Remark 1), the efficiency is reduced because the double precision speed is 1/8 of the single precision speed in GeForce cards with GT200 and GF100 architectures. Secondly, since the register usage and the complexity of the code executed by each thread is higher in this implementation, the CUDA compiler has to store some data into local memory, which also increases the execution time.

Recently, a new simpler numerical scheme for the two-layer shallow water system has been proposed [29]. This scheme can be fully implemented in single precision and therefore, the execution times and speedups are notably improved with

Table 4
Execution times in seconds for all the meshes and programs for the two-layer case.

Mesh size	CPU		GTX 260	
	1 core	4 cores	CUSDP	CUDP
4016	6.86	1.76	2.33	2.29
16040	56.74	14.57	14.46	13.83
64052	496.2	137.1	108.7	106.0
256576	5606.4	1764.5	795.7	887.5
1001898	54154.7	16906.4	5613.9	8209.5

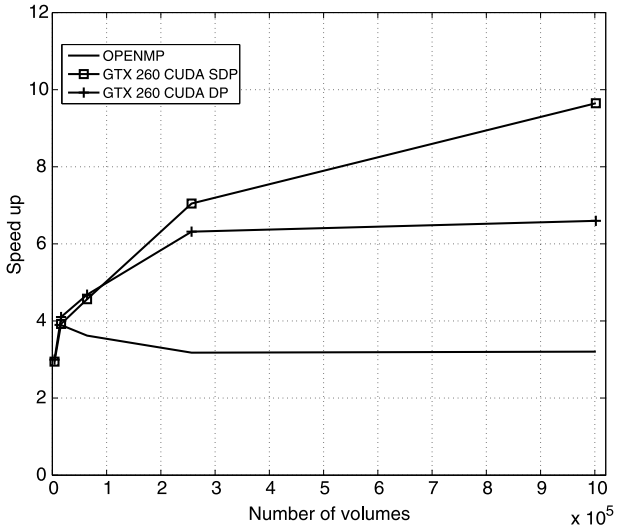


Fig. 13. Speedups obtained in the CUDA and OpenMP implementations in all meshes for the two-layer case.

respect to those presented in this section. Furthermore, the results obtained with this new scheme have similar quality to those obtained with the Roe scheme.

We have also compared the numerical solutions obtained in the monocore and the CUDA programs. The L^1 norm of the difference between the solutions obtained in CPU and GPU at time $t = 1.0$ for all meshes was calculated. The order of magnitude of the L^1 norm using CUSDP varies between 10^{-3} and 10^{-5}, while that of obtained using CUDP varies between 10^{-13} and 10^{-14}, which reflects the different accuracy of the numerical solutions computed on the GPU using both single and double precision, and using only double precision.

8. Conclusions

Different implementations of a path-conservative first and third-order Roe type well-balanced finite volume scheme for the one-layer shallow water system have been performed. Optimization techniques to parallelize efficiently the numerical schemes on CUDA architecture have been considered. Simulations of the one-layer shallow water system carried out on a GeForce GTX 280 and GTX 480 cards using single precision were found to be up to two orders of magnitude faster than a monocore version of the solver for big size uniform problems, one order of magnitude faster than a quadcore implementation based on OpenMP, and also faster than a GPU version based on a graphics-specific language (Cg). The double precision version of the CUDA solver has been 7 times slower than the single precision version for big meshes. In any case, this factor of 7 can be dramatically reduced using a Tesla graphics card based on Fermi architecture, where the number of double precision units is increased. These simulations also show that the numerical solutions obtained with the solver are accurate enough for practical applications, obtaining better accuracy using double precision than using single precision. In the case of two-layer shallow water system the results are not so good and speed up are dramatically reduced, mainly for the use of double precision in the main cores of the algorithm. Nevertheless, in [29] a new scheme for the two-layer shallow water system has been introduced, such as no spectral decomposition of the matrices $\mathcal{A}_{ij}$ are needed, and only some information of the eigenvalues of the system must be provided. The implementation of this scheme is being carried out and the first results are promising: simulations can be performed on single precision and therefore, the speedup is increased, being the quality of the results similar to those obtained with the usual Roe scheme. As further work, we propose to extend the strategy to enable efficient high-order simulations on non-structured meshes.

Acknowledgements

This research has been partially supported by the Spanish Government Research projects MTM2009-11923, TIN2007-29664-E, MTM2008-06349-C03-03, and P06-RNM-01594. The numerical computations have been performed at the Laboratory of Numerical Methods of the University of Málaga.

References

[1] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, C. Parés, A parallel 2D finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows, Comput. Meth. Appl. Mech. Eng. 195 (2006) 2788–2815.
[2] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, C. Parés, Solving shallow-water systems in 2D domains using finite volume methods and multimedia SSE instructions, J. Comput. Appl. Math. 221 (2008) 16–32.
[3] M. Rumpf, R. Strzodka, Graphics processor units: New prospects for parallel computing, Lecture Notes Comput. Sci. Eng. 51 (2006) 89–121.
[4] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing, Proceedings of the IEEE 96 (2008) 879–899.
[5] T.R. Hagen, J.M. Hjelmervik, K.-A. Lie, J.R. Natvig, M.O. Henriksen, Visual simulation of shallow-water waves, Simul. Model. Pract. Theory 13 (2005) 716–726.
[6] M. Lastra, J.M. Mantas, C. Ureña, M.J. Castro, J.A. García-Rodríguez, Simulation of shallow-water systems using graphics processing units, Math. Comput. Simul. 80 (2009) 598–618.
[7] Wen-Yew Liang, Tung-Ju Hsieh, Muhammad Satria, Yang-Lang Chang, Jyh-Perng Fang, Chih-Chia Chen, Chin-Chuan Han, A GPU-based simulation of tsunami propagation and inundation, Lecture Notes in Computer Science 5574 (2009) 593–603.
[8] NVIDIA, CUDA home page, http://www.nvidia.com/object/cuda_home_new.html.
[9] M. de la Asunción, J.M. Mantas, M.J. Castro, Simulation of one-layer shallow water systems on multicore and CUDA architectures, J. Supercomput. (2009), doi:10.1007/s11227-010-0406-2.
[10] M. de la Asunción, J.M. Mantas, M.J. Castro, Programming CUDA-based GPUs to simulate two-layer shallow water flows, Euro-Par 2010, Ischia, Italy.
[11] M.J. Castro, S. Ortega, M. de la Asunción, J.M. Mantas, On the benefits of using GPUs to simulate shallow flows with finite volume schemes, Boletín de la Sociedad Española de Matemática Aplicada 50 (2010) 27–44.
[12] A.R. Brodtkorb, T.R. Hagen, K.-A. Lie, J.R. Natvig, Simulation and visualization of the Saint-Venant system using GPUs, Computing and Visualization in Science, Special issue on Hot Topics in Computational Engineering (2010), doi:10.1007/s00791-010-0149-x, in press.
[13] M.J. Castro, E.D. Fernández, A.M. Ferreiro, A. García, C. Parés, High order extension of Roe schemes for two-dimensional nonconservative hyperbolic systems, J. Sci. Comput. 39 (2009) 67–114.
[14] G. Dal Maso, P.G. LeFloch, F. Murat, Definition and weak stability of nonconservative products, J. Math. Pures Appl. 74 (1995) 483–548.
[15] A.I. Volpert, Spaces BV quasilinear equations, Math. USSR Sbornik 73 (1967) 255–302.
[16] A. Harten, J.M. Hyman, Self-adjusting grid methods for one-dimensional hyperbolic conservation laws, J. Comp. Phys. 50 (1983) 235–269.
[17] M.J. Castro, P.G. LeFloch, M.L. Muñoz, C. Parés, Why many theories of shock waves are necessary: Convergence error in formally path-consistent schemes, Jour. Comp. Phys. 3227 (2008) 8107–8129.
[18] C. Parés, M.L. Muñoz Ruíz, On some difficulties of the numerical approximation of nonconservative hyperbolic systems, Boletín SEMA 47 (2009) 23–52.
[19] T.Y. Hou, P.G. LeFloch, Why nonconservative schemes converge to wrong solutions: error analysis, Math. Comput. 62 (1994) 497–530.
[20] M.L. Muñoz, C. Parés, On the convergence and well-balanced property of path-conservative numerical schemes for systems of balance laws, J. Sci. Comp. (2010), doi:10.1007/s10915-010-9425-7, in press.
[21] S. Noelle, N. Pankratz, G. Puppo, J. Natvig, Well-balanced finite volume schemes of arbitrary order of accuracy for shallow water flows, J. Comput. Phys. 213 (2006) 474–499.
[22] G. Walz, Romberg type cubature over arbitrary triangles, Mannheimer Mathem, Manuskripte Nr. 225, Mannhein, 1997.
[23] C.-W. Shu, S. Osher, Efficient implementation of essentially non-oscillatory shock capturing schemes, J. Comput. Phys. 77 (1998) 439–471.
[24] J.M. Gallardo, S. Ortega, M. de la Asunción, J.M. Mantas, Two-dimensional compact third-order polynomial reconstructions. Solving nonconservative hyperbolic systems using GPUs, J. Sci. Comput. (2010), submitted for publication.
[25] NVIDIA, NVIDIA CUDA Programming Guide Version 3.0, 2010, http://developer.nvidia.com/object/cuda_3_0_downloads.html.
[26] Eigen 2.0.15, http://eigen.tuxfamily.org.
[27] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programing, The MIT Press, 2007.
[28] EISPACK http://www.netlib.org/eispack.
[29] E.D. Fernández Nieto, M.J. Castro, C. Parés, IFCP Riemann solver for the two-layer shallow-water system, J. Sci. Comput. (2010), submitted for publication.