Parallelization of Shallow Water Simulations on Current Multi-threaded Systems

J. Lobeiras (jacobo.lobeiras@udc.es) *
M. Viñas (moises.vinas@udc.es) *
M. Amor (margarita.amor@udc.es) *
B.B. Fraguela (basilio.fraguela@udc.es) *
M. Arenaz (manuel.arenaz@udc.es) †
J.A. García (jagrodriguez@udc.es) ‡
M.J. Castro (castro@anamat.cie.uma.es) §

September 24, 2012

Abstract

In this work, several parallel implementations of a numerical model of pollutant transport on a shallow water system are presented. These parallel implementations are developed in two phases. First, the sequential code is rewritten to exploit the stream programming model. And second, the streamed code is targeted for current multi-threaded systems, in particular, multi-core *CPUs* and modern *GPUs*. The performance is evaluated on a multi-core *CPU* using *OpenMP*, and on a *GPU* using the streaming-oriented programming language *Brook+*, as well as the standard language for heterogeneous systems *OpenCL*.

Keywords: Shallow water, pollutant transport, stream programming, compiler parallelizing transformations, GPU, OpenMP, Brook+, OpenCL.

*Computer Architecture Group (GAC), Univ. of A Coruña (UDC), A Coruña, Spain
†Univ. of A Coruña (UDC), A Coruña, Spain
‡M2NICA group, Univ. of A Coruña (UDC), A Coruña, Spain
§EDANYA group, Univ. of Málaga (UMA), Málaga, Spain

1 Introduction

Shallow water systems describe the evolution of an incompressible fluid in response to gravitational accelerations, where the vertical flow is small compared to the horizontal flow. These systems have many applications, enabling the simulation of rivers, canals, coastal hydrodynamics or dam-break problems, among others. In particular, the transport of pollutant in a fluid, that is modelled by a transport equation, has particular relevance in many ecological and environmental studies. This paper uses a mathematical model that consists in the coupling of a shallow water system and a transport equation. These coupled equations constitute a hyperbolic system of conservation laws with source terms, that can be discretized using finite volume schemes (R.J. LeVeque, 2002; E.F. Toro, 2001).

Finite volume schemes solve the integral form of the shallow water equations in computational cells of a geometrical mesh that describes the computational domain. Some main benefits of using explicit finite volume schemes are observed. First, mass and momentum are conserved in each cell, even in the presence of flow discontinuities. A good approximation of fast waves as moving shocks or wet-dry fronts appears in fluid or coastal hydraulics. Furthermore, much reduced memory overheads are involved, as complex iterative matrix solvers are not required. Finally, explicit finite volume schemes are easy to implement in multi-core and many-core systems as the most intensive computational part of the algorithm consists of a set of operations that can be performed independently (and thus asynchronously) at each edge of the mesh. This set of operations can be identified with a lightweight computational kernel, which is invoked a large number of times for big meshes, and thus the algorithm fits perfectly a stream programming model.

The simulations of these problems have very large computing requirements which grow with the size of the space and time dimensions of the domain. For example, in the simulation of marine systems, the spatial domain can have many kilometers and the time integration of the problem can last several weeks or even months. Precise simulations over large detailed terrains require big meshes that usually result in prohibitive execution times.

Thus, due to the interest of this kind of problems and its high computational demands together with the fact that explicit FV solvers fit well with the streaming programming model, several parallel implementations have been proposed on a wide variety of platforms, such as computer clusters using *MPI* (M.J. Castro et al., 2006), a version combining *MPI* and *SSE* (*Streaming SIMD Extensions*) instructions (M.J. Castro et al., 2008a) and other generic multi-platform implementations like (D. van Dyk et al., 2009). Despite these efforts, the increasing computing power required by the most complex simulations motivated the development of *GPU* (*Graphics Processing Unit*) solvers (T. Runar et al., 2006; M. Lastra et al., 2009; T.R. Hagen et al., 2007) based on the first generation of *GPU* programming languages like *Cg* or *GLSL*. The rapidly increasing computational power and low cost of *GPUs* and the advances in *GPU* high-level programming languages motivated the development of new parallel versions for modern *GPUs*. Examples of *CUDA* implementations are a one-layer simulator (M. Geveler et al., 2010; D. Ribbrock et al., 2010), a multi-*GPU* version (M.L. Sætra and A.R. Brodtkorb, 2012) or high order implementations (A.R. Brodtkorb et al., 2012; J.M. Gallardo et al., 2011). The parallel implementations mentioned above do not handle pollutant transport problems. Even if single species transport does not introduce any mathematical difficulties, we have decided to consider SWE together with pollutant transport equation as this system is the basis of more complicated models as turbidity current system presented in (T. Morales de Luna et al., 2009). Turbidity currents are of great interest as those have a big impact on the morphology of the continental shelves and ocean basins. Thus, the scheme presented in this paper can be easily adapted to solve 2D turbidity currents following the aforementioned work. A direct implementation for pollutant transport simulation on *CUDA GPUs* was presented in (M. Viñas et al., 2011).

This paper proposes a parallel shallow water simulator that solves a broad variety of problems, even with pollutant transport and the presence of wet-dry fronts in emerging bottom situations, and which runs very efficiently on current multi-threaded architectures. Our approach first applies generic parallelizing transformations to rewrite the sequential code following the stream programming model. In this paradigm the same function or *streaming kernel* is applied to a set of inputs in parallel, producing another set of outputs. There should be no data dependencies among the threads nor overlapping between the input and the output data to prevent race conditions. This model is designed to encourage and exploit a high degree of parallelism without significant compiler effort, offering flexibility to exploit current *GPUs* and multi-core *CPUs*. Then, the streaming sequential version is fine-tuned to exploit the hardware characteristics of multi-core *CPUs* using *OpenMP* (R. Chandra et al., 2001) and of modern *GPUs* using *Brook+* (AMD, 2009) and *OpenCL* (Khr, 2011). Our two-phase parallelization approach contributes to reduce development time as well as maintenance costs. This paper shows that shallow water problems are well suited for the stream paradigm, and that it is possible to take advantage efficiently of the stream programming model in both modern *GPUs* and multi-core *CPUs*. The resulting implementations achieve very good scalability on *CPUs* using *OpenMP* and excellent performance on *GPUs* using either *Brook+* or *OpenCL*, which enables really large simulations even when dealing with pollutant transport problems and wet-dry zones on very complex terrains.

The outline of the article is as follows. Section 2 describes the mathematical model, which in our case consists in the coupling of a shallow water system and a transport equation in a bidimensional domain. Section 3 presents the numerical scheme that approximates the solution of the mathematical model. Section 4 introduces the structure of the sequential numerical algorithm. Section 5 presents our two-phase approach to develop three efficient parallel versions, on multi-core *CPUs* with *OpenMP* and on *GPUs* with *Brook+* and *OpenCL*. Section 6 presents experimental results for an academic 2D dam-break problem to asses the correctness and the accuracy of the parallel implementation. It also presents results for a realistic domain, the Ría de Arousa located in Galicia (North-West Spanish region), comparing the performance and scalability of the *CPU/OpenMP*, *GPU/Brook+*, and *GPU/OpenCL* implementations. Finally, Section 7 presents conclusions and future work.

2 Coupled model: 2D shallow water equations with pollutant transport

A pollutant transport model consists in the coupling of a fluid model and a transport equation. In this work, to model the fluid dynamics we consider the bidimensional shallow water equations, which describe the evolution of a fluid over a bottom, where the thickness and the vertical flow is small compared to the horizontal flow:

$$\begin{cases} \dfrac{\partial h}{\partial t} + \dfrac{\partial q_x}{\partial x} + \dfrac{\partial q_y}{\partial x} = 0, \\ \dfrac{\partial q_x}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x^2}{h} + \dfrac{1}{2}gh^2\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_x q_y}{h}\right) = gh\dfrac{\partial H}{\partial x} + ghS_{f,x}, \\ \dfrac{\partial q_y}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x q_y}{h}\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_y^2}{h} + \dfrac{1}{2}gh^2\right) = gh\dfrac{\partial H}{\partial y} + ghS_{f,y}. \end{cases} \tag{1}$$

The unknowns of the problem are the vertically averaged height of the water column $h(\boldsymbol{x}, t)$ and the flux $\boldsymbol{q}(\boldsymbol{x}, t) = (q_x(\boldsymbol{x}, t), q_y(\boldsymbol{x}, t)) = h(\boldsymbol{x}, t) \cdot \boldsymbol{u}(\boldsymbol{x}, t)$, where

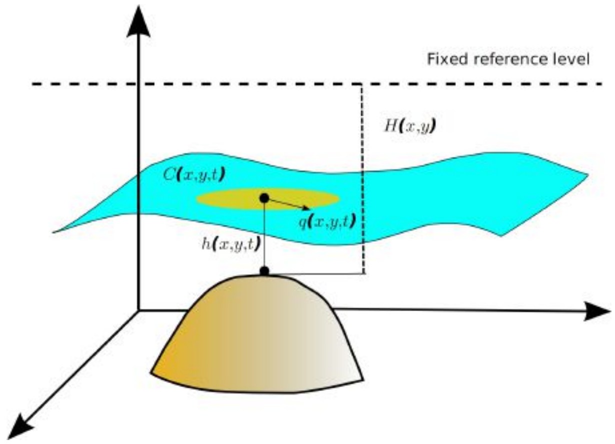


Figure 1: Sketch: pollutant transport.

$\boldsymbol{u}(\boldsymbol{x},t) = (u_x(\boldsymbol{x},t), u_y(\boldsymbol{x},t))$ is the vertical averaged velocity of the fluid, at each point $\boldsymbol{x} = (x,y)$ of the computational domain and at time t. $H(\boldsymbol{x})$ is the function that describes the bottom bathymetry, measured from a fixed reference level (see Figure 1), and g is the gravitational constant.

The friction forces are given by a Manning Law:

$$S_{f,x} = n^2 \frac{u_x \|\boldsymbol{u}\|}{h^{1/3}}, \quad S_{f,y} = n^2 \frac{u_y \|\boldsymbol{u}\|}{h^{1/3}}, \tag{2}$$

where n is the bed roughness coefficient.

The pollutant transport equation is given by:

$$\frac{\partial(hC)}{\partial t} + \frac{\partial(q_x C)}{\partial x} + \frac{\partial(q_y C)}{\partial y} = 0, \tag{3}$$

where $C(\boldsymbol{x},t)$ is the pollutant concentration.

The system given by Equations (1) and (3) can be written as a *system of conservation laws with source terms*:

$$\frac{\partial W}{\partial t} + \frac{\partial}{\partial x}F_1(W) + \frac{\partial}{\partial y}F_2(W) = S_1(W)\frac{\partial}{\partial x}H(\boldsymbol{x}) + S_2(W)\frac{\partial}{\partial y}H(\boldsymbol{y}) + \boldsymbol{S_f}, \tag{4}$$

where W is the vector of unknowns:

$$W = \begin{bmatrix} h \\ q_x \\ q_y \\ hC \end{bmatrix}, \tag{5}$$

where $h(\boldsymbol{x},t)C(\boldsymbol{x},t)$ is the amount of pollutant dissolved in the fluid, and

$$F_1(W) = \begin{bmatrix} q_x \\ \dfrac{q_x^2}{h} + \dfrac{1}{2}gh^2 \\ \dfrac{q_x q_y}{h} \\ q_x C \end{bmatrix}, \; F_2(W) = \begin{bmatrix} q_y \\ \dfrac{q_x q_y}{h} \\ \dfrac{q_y^2}{h} + \dfrac{1}{2}gh^2 \\ q_y C \end{bmatrix},$$

$$S_1(W) = \begin{bmatrix} 0 \\ gh \\ 0 \\ 0 \end{bmatrix}, \quad S_2(W) = \begin{bmatrix} 0 \\ 0 \\ gh \\ 0 \end{bmatrix}, \tag{6}$$

and

$$\boldsymbol{S_f} = \begin{bmatrix} 0 \\ ghS_{f,x} \\ ghS_{f,y} \\ 0 \end{bmatrix}. \tag{7}$$

System (4) can be written in a more compact form:

$$\frac{\partial W}{\partial t}(\boldsymbol{x},t) + \mathrm{div}\boldsymbol{F}(W) = \boldsymbol{S}(W)\cdot\nabla H(\boldsymbol{x}) + \boldsymbol{S_f}, \tag{8}$$

where $\boldsymbol{F} = (F_1, F_2)$ is the flux function and $\boldsymbol{S}(W) = (S_1(W), S_2(W))$.

Given an unitary vector $\boldsymbol{\eta} = (\eta_x, \eta_y)$, we define the matrix

$$A(W,\boldsymbol{\eta}) = A_1(W)\eta_x + A_2(W)\eta_y, \tag{9}$$

where

$$A_1(W) = \frac{\partial}{\partial W}F_1(W), \quad A_2(W) = \frac{\partial}{\partial W}F_2(W) \tag{10}$$

are the jacobian matrices of $F_1(W)$ and $F_2(W)$, respectively.

System (8) is hyperbolic if $h(\boldsymbol{x},t) > 0$. Effectively, $A(W,\boldsymbol{\eta})$ is diagonalizable and the eigenvalues of $A(W,\boldsymbol{\eta})$ are

$$\lambda_1 = \boldsymbol{u}\cdot\boldsymbol{\eta},\; \lambda_2 = \boldsymbol{u}\cdot\boldsymbol{\eta} - \sqrt{gh},\; \lambda_3 = \boldsymbol{u}\cdot\boldsymbol{\eta} + \sqrt{gh},\; \lambda_4 = \boldsymbol{u}\cdot\boldsymbol{\eta}. \tag{11}$$

3 Finite volume numerical scheme.

In this section we briefly describe the finite volume scheme that we use to discretize the Equation (8). More details can be found in (M.J. Castro et al., 2006, 2008b, 2009).

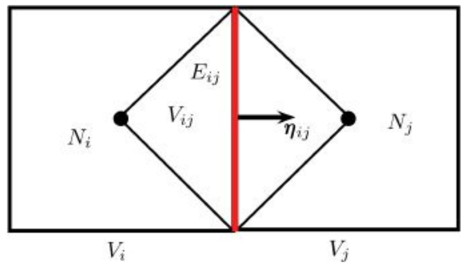


Figure 2: Finite volume: structured mesh.

Let us remark that the term $\boldsymbol{S_f}$ is discretized in a semi-implicit way as detailed in (M.J. Castro et al., 2008b), thus in what follows we focus on the discretization of Equation (8) where $\boldsymbol{S_f}$ is supposed to be zero.

To discretize the Equation (8), we split the computational domain in cells or control volumes, $V_i \subset \mathbb{R}^2,\ i = 1, \ldots, L$. In our case we will consider a structured mesh given by squares. We will use the following notation: given a finite volume V_i, $\boldsymbol{x}_i$ is its center, $|V_i|$ its area, $\mathcal{N}_i$ is the set of indexes j such that V_j is the neighbor of V_i, E_{ij} is the edge shared by two neighbor cells V_i and V_j and $|E_{ij}|$ is its length, and $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unitary vectorial normal to edge E_{ij} and that points towards the cell V_j (see Figure 2). Finally, we call V_{ij} the triangular subcell with one edge given by E_{ij} and the opposite vertex given by $\boldsymbol{x}_i$ (see Figure 2).

In finite volume schemes, constant approximations of the solution at each cell are computed. More precisely, if $W(\boldsymbol{x}, t)$ is the exact solution at point $\boldsymbol{x}$ and at time t, we will denote by ${W_i}^n$ an approximation of the average of the solution on the volume V_i at time t^n,

$$ {W_i}^n \simeq \frac{1}{|V_i|} \int_{V_i} W(\boldsymbol{x}, t^n) d\boldsymbol{x}. \tag{12} $$

Integrating the Equation (8) over each finite volume V_i

$$ \frac{\partial}{\partial t} \int_{V_i} W(\boldsymbol{x}, t)\, dV + \int_{V_i} (\mathrm{div}\mathbf{F}(W))\, dV = \int_{V_i} \boldsymbol{S}(W) \cdot \nabla H(\boldsymbol{x})\, dV. \tag{13} $$

Dividing by $|V_i|$ and applying the Divergence Theorem:

$$ \begin{aligned} &\frac{\partial}{\partial t} \left(\frac{1}{|V_i|} \int_{V_i} W(\boldsymbol{x}, t)\, dV \right) = \\ &\quad -\frac{1}{|V_i|} \left(\sum_{j \in \mathcal{N}_i} \int_{E_{ij}} \mathbf{F}(W) \cdot \boldsymbol{\eta}_{ij}\, d\gamma - \int_{V_i} \boldsymbol{S}(W) \cdot \nabla H(\boldsymbol{x})\, dV \right). \end{aligned} \tag{14} $$

To discretize Equation (14) we will use the finite volume numerical scheme presented in (M.J. Castro et al., 2008b). Once the approximation of W_i is known at time t^n, W_i^n, the approximation at time t^{n+1} is given by:

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

with

$$\begin{aligned} \mathcal{F}_{ij}^-(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) = \mathcal{P}_{ij}^n \Big(\boldsymbol{F}(W_j^n) \cdot \boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n) \cdot \boldsymbol{\eta}_{ij} - \boldsymbol{S}_{ij}^n \Big) \\ - \frac{\boldsymbol{F}(W_j^n) \cdot \boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n) \cdot \boldsymbol{\eta}_{ij}}{2} + \boldsymbol{F}_\alpha(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) - \boldsymbol{S}_{\alpha,ij}^n, \end{aligned} \tag{16}$$

where the projection matrix, $\mathcal{P}_{ij}$, is given by:

$$\mathcal{P}_{ij}^n = \frac{1}{2} K_{ij}^n \Big(I - \operatorname{sgn}(D_{ij}^n) \Big) (K_{ij}^n)^{-1}, \tag{17}$$

being I the identity matrix and K_{ij}^n the matrix whose columns are the eigenvectors related to the Roe matrix A_{ij}^n given by

$$A_{ij}^n = A(W_{ij}^n, \boldsymbol{\eta}_{ij}) = A_1(W_{ij}^n)\eta_{ij,x} + A_2(W_{ij}^n)\eta_{ij,y}, \tag{18}$$

where

$$W_{ij}^n = \begin{bmatrix} h_{ij}^n \\ h_{ij}^n u_{ij,x}^n \\ h_{ij}^n u_{ij,y}^n \\ h_{ij}^n C_{ij}^n \end{bmatrix}, \tag{19}$$

is the intermediate Roe's state, which is the state that satisfies the equation

$$\boldsymbol{F}(W_j^n) \cdot \boldsymbol{\eta}_{ij} - \boldsymbol{F}(W_i^n) \cdot \boldsymbol{\eta}_{ij} = A_{ij}^n (W_j^n - W_i^n) \tag{20}$$

and it is given by:

$$h_{ij}^n = \frac{h_i^n + h_j^n}{2}, \tag{21}$$

$$u_{ij,l}^n = \frac{\sqrt{h_i^n} u_{i,l}^n + \sqrt{h_j^n} u_{j,l}^n}{\sqrt{h_i^n} + \sqrt{h_j^n}}, \ l = x, y, \tag{22}$$

$$C_{ij}^n = \frac{\sqrt{h_i^n} C_i^n + \sqrt{h_j^n} C_j^n}{\sqrt{h_i^n} + \sqrt{h_j^n}}. \tag{23}$$

D_{ij}^n the diagonal matrix whose elements are the eigenvalues of A_{ij}^n that are given by:

$$\begin{cases} \lambda_{ij,1} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij}, \\ \lambda_{ij,2} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij} - \sqrt{g h_{ij}^n}, \\ \lambda_{ij,3} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij} + \sqrt{g h_{ij}^n}, \\ \lambda_{ij,4} = \boldsymbol{u}_{ij}^n \cdot \boldsymbol{\eta}_{ij}, \end{cases} \tag{24}$$

and

$$\mathrm{sgn}\ D_{ij}^n = \begin{bmatrix} \mathrm{sgn}\ \lambda_{ij,1} & & & \\ & \mathrm{sgn}\ \lambda_{ij,2} & & \\ & & \mathrm{sgn}\ \lambda_{ij,3} & \\ & & & \mathrm{sgn}\ \lambda_{ij,4} \end{bmatrix}. \tag{25}$$

The term $\boldsymbol{S}_{ij}^n$ is given by

$$\boldsymbol{S}_{ij}^n = \begin{bmatrix} 0 \\ g h_{ij}^n (H_j - H_i) \eta_{ij,x} \\ g h_{ij}^n (H_j - H_i) \eta_{ij,y} \\ 0 \end{bmatrix}. \tag{26}$$

$$\boldsymbol{F}_\alpha(W_i^n, W_j^n, \boldsymbol{\eta}_{ij}) = \frac{\boldsymbol{F}(W_{(1-\alpha)i+\alpha j}) \cdot \boldsymbol{\eta}_{ij} + \boldsymbol{F}(W_{\alpha i+(1-\alpha)j}) \cdot \boldsymbol{\eta}_{ij}}{2}, \tag{27}$$

where we denote:

$$W_{(1-\alpha)i+\alpha_j} = \begin{bmatrix} h_{(1-\alpha)i+\alpha_j} \\ (q_x)_{(1-\alpha)i+\alpha_j} \\ (q_y)_{(1-\alpha)i+\alpha_j} \\ hC_{(1-\alpha)i+\alpha_j} \end{bmatrix} = (1-\alpha) W_i^n + \alpha W_j^n, \quad \alpha \in [0,1], \tag{28}$$

a convex combination of W_i^n and W_j^n, and finally,

$$\boldsymbol{S}_{\alpha,ij}^n = \begin{bmatrix} 0 \\ \frac{g}{2} \left(\frac{h_{(1-\alpha)i+\alpha j} + h_i^n}{2} (H_{(1-\alpha)i+\alpha j} - H_i) + \frac{h_{\alpha i+(1-\alpha)j} + h_j^n}{2} (H_{\alpha i+(1-\alpha)j} - H_j) \right) \eta_{ij,x} \\ \frac{g}{2} \left(\frac{h_{(1-\alpha)i+\alpha j} + h_i^n}{2} (H_{(1-\alpha)i+\alpha j} - H_i) + \frac{h_{\alpha i+(1-\alpha)j} + h_j^n}{2} (H_{\alpha i+(1-\alpha)j} - H_j) \right) \eta_{ij,y} \\ 0 \end{bmatrix}, \tag{29}$$

where

$$H_{\alpha i+(1-\alpha)j} = \alpha H_i + (1-\alpha)H_j, \tag{30}$$

is again a convex combination of H_i and H_j.

The Equations (27) and (29) are used to avoid entropy corrections needed by the Roe scheme in critical points (see (M.J. Castro et al., 2008b)). The authors propose different values of the parameter α. In practice, the value $\alpha = 1/8$ gives good results (see (M.J. Castro et al., 2008b)), so here we take $\alpha = 1/8$. Note that in the case $\alpha = 0$ we obtain the usual Roe Scheme (see (M.J. Castro et al., 2008b)).

The previous numerical scheme is exactly well-balanced for the stationary solution corresponding to water at rest (see (M.J. Castro et al., 2008b)) and linearly L^∞ under the usual CFL condition:

$$\Delta t = \min_{i=1,\ldots,L} \left\{ \frac{\sum_{j\in\mathcal{N}_i} |E_{ij}| \, \|D_{ij}^n\|_\infty}{2\gamma|V_i|} \right\} \tag{31}$$

where γ, $0 < \gamma \leq 1$, is the CFL parameter and $\|D_{ij}^n\|_\infty$ is the infinite norm of the matrix D_{ij}^n, that is, the maximum eigenvalue of the matrix A_{ij}^n.

The resulting time step can be small, which gives rise to a large number of time steps for simulations that occur on large time scales, which is the case for many geophysical flow problems. Thus, from the computational point of view, the solution of the problem is reduced to a huge number of matrix operations and vectors of size 4×4.

Finally, let us recall that the finite volume scheme described in this section is of first order. High order schemes have been implemented in *CPU* (M.J. Castro et al., 2009) and in *GPUs* (A.R. Brodtkorb et al., 2012; J.M. Gallardo et al., 2011) and they provide very good results in academic examples. Nevertheless, the extension of those schemes to simulate real flows with real bathymetries is not a simple task and sometimes, they produce inaccurate results in wet-dry fronts. Let us also remark that this scheme is a generalization of Roe scheme, which gives very precise results and, moreover, it can approximate stationary regular solutions up to second order (see Theorem 10 in (M.J. Castro et al., 2009)).

3.1 Wet-dry fronts

One of the main difficulties that can appear in practical applications is the presence of wet-dry fronts. These fronts develop when, due to the initial conditions or as a consequence of the fluid motion, the thickness of the layer vanishes. These situations arise very frequently in practical applications such as flood waves, dam-breaks or coastal tidal currents. We handle this situation in two ways. First, we compute the velocities and concentrations as follows (A. Kurganov and G. Petrova, 2007):

$$lu_{i,x} = \frac{\sqrt{2}h_i q_{i,x}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{32}$$

$$u_{i,y} = \frac{\sqrt{2}h_i q_{i,y}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{33}$$

$$C_i = \frac{\sqrt{2}h_i q_{i,C}}{\sqrt{h_i^4 + max(h_i, \varepsilon)^4}}, \tag{34}$$

where $\varepsilon = 10^{-6}$ is the single precision limit. In practical situations this value gives good results.

Second, if the thickness of the layer of fluid becomes tiny at both cells V_i and V_j, that is $h_i, h_j < h_{eps} = 10^{-4}$, then the fourth component of the numerical flux $\mathcal{F}_{ij}^{-}(W_i^n, W_j^n, \boldsymbol{\eta}_{ij})$ is defined as follows:

$$\mathcal{F}_{ij}^{-}{}_{[4]} = \begin{cases} \mathcal{F}_{ij}^{-}{}_{[1]} \cdot C_j & \text{if } \boldsymbol{u}_{ij} \cdot \boldsymbol{\eta}_{ij} < 0, \\ \mathcal{F}_{ij}^{-}{}_{[1]} \cdot C_i & \text{if } \boldsymbol{u}_{ij} \cdot \boldsymbol{\eta}_{ij} > 0, \end{cases} \tag{35}$$

where $\mathcal{F}_{ij}^{-}{}_{[l]}$, denotes the l-th component of the vector $\mathcal{F}_{ij}^{-}$. This value of h_{eps} has been chosen such as gives the best results in the numerical experiments we have performed.

It must be remarked that the numerical scheme described in the previous section corresponds to the case where the fluid occupies the whole domain. If this numerical scheme is applied without any modification to a case with wet-dry fronts (situations with emerging bottom topography), the results obtained have spurious values. In those cases it is necessary to modify the scheme, as is proposed in (M.J. Castro et al., 2008b). This modification allows to balance the fluxes against the driving forces so that the non-physical pressure forces disappear in the case of bottom emerging topographies.

Finally, let us remark that in order to provide numerical simulations in real domains, friction terms are very important to reproduce the correct position of wet-dry fronts. Moreover, the semi-implicit way of discretizing the friction terms enforces the numerical stability of the scheme in areas where h is small (see (M.J. Castro et al., 2008b) for more details).

4 Structure of the sequential algorithm

The edge-based algorithm that approximates the solution of the numerical scheme of the coupled system given by Equation (4) is shown in Figure 3(a). It mainly consists of a loop that performs a simulation through time. In each time step, the amount of flow that crosses through each edge is calculated in order to compute the flow data corresponding to every finite volume of the mesh. For each edge of the mesh, this edge-driven algorithm performs a huge number of small vector and matrix operations (e.g., product or inverse) to solve the equations of the coupled system. Each time iteration is divided into three stages:

① Compute the numerical fluxes ΔM and the time step Δt for each volume v (see Stage ①), where $\Delta M[V_i] = \sum_{j\in\mathcal{N}_i} \mathcal{F}_{ij}^{-}(W_i, W_j, \eta_{ij})$. For each edge a, the amount of fluid and pollutant that crosses the edge towards the neighbor volume on the left, $\Delta M[left(a)]$, is computed. Furthermore, the contribution to the neighbor on the right, $\Delta M[right(a)]$), is also computed. At the beginning of this stage for each volume v, $\Delta M[v]$ and $\Delta t[v]$ are initialized to the value zero (flow information $\Delta M[v]$ has four components: the water column height, the volume flow in the x and y coordinates, and the pollutant concentration). Upon the completion of this stage, every finite volume will have received the contributions from all of its four neighbor edges. The time step Δt of each volume is computed in a similar manner. This stage corresponds to the expression inside summation of Equation (15).

② Compute the global time step $\Delta tGlobal$ (see Stage ②) as the minimum of the local time steps Δt computed for each volume in Stage ①. This stage corresponds to the minimum computed in Equation (31).

③ Compute the simulated flow data M for each volume (see Stage ③). This is achieved by updating in each volume the pollutant density and fluid data using ΔM from Stage ① and $\Delta tGlobal$ from Stage ②. This stage corresponds to the right-hand side of Equation (15), after computing the summation.

Stage ① is the most computationally intensive part, because the large number of small vector and matrix operations are numerically intensive. This way, for an example mesh of 4000×2666 volumes, a profiling execution of the code reveals that about 75% of the total CPU execution time is consumed by the first stage.

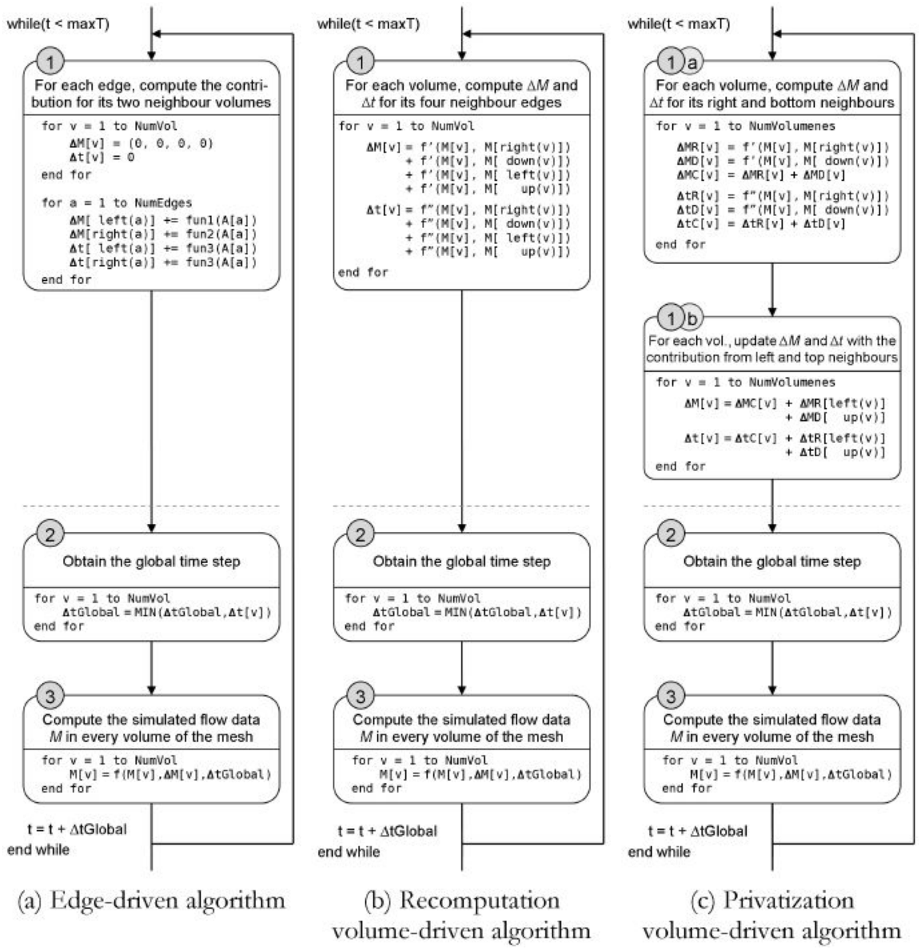


Figure 3: Diagram comparing the studied implementations.

Consequently, the following section of the paper describes our efficient and cost-effective parallel implementation of the algorithm, with special attention to Stage ① and its coupling with the other stages of the algorithm.

5 Parallelization on multi-threaded systems

The parallelization of large real applications is a complex process. Many techniques and tools have been developed to assist programmers in this manual process. In this work, we have used an analysis based on domain-independent kernels (M. Arenaz et al., 2008), which have been successfully applied to the parallelization of algorithms and full-scale applications for *CPUs* and *GPUs* (M. Arenaz et al., 2004; J. Setoain et al., 2008).

The rest of the section is organized as follows. Section 5.1 deals with the optimization of the sequential code. Section 5.2 addresses the construction of the streamed version. Section 5.3, Section 5.4, and Section 5.5 describe the efficient parallel implementations of the streamed version for multi-core *CPUs* with *OpenMP*, for *GPUs* with *Brook+*, and for *GPUs* with *OpenCL*, respectively.

5.1 Optimization of the sequential code

The first step of the development process was to optimize our initial sequential implementation, whose numerical scheme was described in Section 3. In our problem there are zones in the mesh that are free of fluid (the first component of W in Equation (5) is zero), either because they are dry terrain areas where the water will never reach, or because some volumes could become dry during the simulation due to tides or water natural flow. In particular, if a volume is dry and its four neighbor volumes are dry as well, it will receive no flow contribution. Thus, the first optimization applied to the sequential code is to avoid the computation of empty volumes by checking if the surrounding volumes are dry as well. This is specially useful if the simulated environment has mountains or large elevated terrain zones, where the water will never reach. On *CPUs* this optimization is specially relevant due to their more limited computing power.

Other well-known code transformations have been manually applied in order to make the sequential code amenable to the compiler and to the stream programming model. Thus, loop-invariant elimination, loop unrolling and common subexpression elimination were applied. Examples of other transformations are computing the equivalent expression of a particular inverse matrix (like the inverse of K in Equation (17)), or simplifying expressions containing edge normal and unitary vectors. The use of symbolic algebraic manipulation software was shown to be helpful in situations where complex expressions make manual optimizations very complex and the compiler automatic optimization does not work as expected.

Current *CPU* architectures support special instructions to speed up common operations involving small vectors, like the *SSE* instructions (*Streaming SIMD Extensions*) in the *x86* architecture or the *AltiVec* extensions in the *Power* and *Cell* architectures. These instructions can be used to perform the same operation in parallel over a small set of elements (typically up to four scalar values, but this depends on the hardware and the data types). *Visual C++ 2008*, which is the compiler used in this work, has basic support to automatically generate vectorized code without the explicit usage of *SIMD* instructions by the programmer. One example where *SIMD* instructions can have a great benefit is in Equation (17) (used in Stage ②) because this equation involves many small matrix and vector operations. According to our experiments, with the autovectorization feature of the *Microsoft Visual C++ 2008* compiler, our implementation achieves about a 85% performance increase.

5.2 Construction of the streamed code

As shown in Figure 3(a), the sequential algorithm consists of three stages. As each stage depends on the results of the previous one, they cannot be reordered for parallel execution. As discussed in Section 4, Stage ① is the most time-consuming part of the algorithm. It consists of a loop that traverses the edges of the mesh so that each edge a writes its contribution to its two neighbor volumes $left(a)$ and $right(a)$. As a result, the concurrent execution of this loop may cause write conflicts among different iterations, so the value of the sum of the edge contributions (which will be stored in ΔM and Δt) would be undefined. In the literature about parallel programming, there exist three main solutions to this problem:

Synchronization-based solution. Synchronizations are used to protect write operations to shared variables by several threads. In this model, the set of edges of Stage ① is divided among the threads, sharing the variables ΔM and Δt. Conflicts are avoided by executing write operations through atomic instructions or critical sections. The implementation of this solution is very simple. However, it may affect performance seriously, so its application must be carefully studied for each application on each target architecture.

Recomputation-based solution. Recomputation is used to avoid communications and synchronization among the threads. In this solution, the sets of elements ΔM and Δt are divided among the threads creating blocks of adjacent volumes. Each thread is responsible for computing all the data associated to its block of volumes. This way, this implementation would be volume-driven, rather than edge-driven because each thread would iterate on the volumes assigned to it. Note that using this approach, the edges common to the volumes of two threads (located at the borders of the block) are processed twice, once for each neighbor volume. As a result, some redundant calculations are computed. There is a particular case when the size of the block is equal to a single volume, each thread has to compute the flow associated to the four edges of the volume. In this case, note that each edge is processed twice. The contribution that volume v_m does to volume v_n takes the same value (though distinct sign) as the contribution of volume v_n to volume v_m. However this contribution is recalculated when volume v_m computes the contribution from its neighbors, thus half of the computations will be redundant.

Privatization-based solution. Privatization is used to avoid write conflicts and minimize synchronization. In this two-stage strategy each thread only computes the contributions from its right and bottom neighbors, and each thread owns a private copy ΔMC and ΔtC. In the first stage, the threads run in a conflict-free manner by computing partial results in ΔMC and ΔtC, and writing the partial results in two communication buffers (for the right and bottom edges respectively). In the second stage, the contributions stored in the communication buffers are merged safely. Each thread reads data from the communication buffers of other threads in order to compute the final results ΔM and Δt.

The first streamed code proposed in this paper is built by rewriting Stage ① following the recomputation-based solution, where all threads run concurrently in a conflict-free manner by writing on different locations of ΔM and Δt. Notice that this involves changing from an edge-driven approach to a volume-driven one, giving place to the algorithm in Figure 3(b).

Stage ② computes the minimum time step $\Delta tGlobal$ of the Δt structure, which is a type of reduction operation. Reductions are collective operations that obtain a single value from several elements. If the reduction function is associative and commutative, the reduction may be rewritten for parallel execution by applying the privatization-based solution. Thus, the reduction variable is privatized to store thread-local partial results, which are later merged safely to compute global results with the appropriate synchronization mechanisms. Finally, note that parallel reductions are very common, so they are natively supported in many programming languages.

Stage ③ updates the simulated flow data M in every volume of the mesh using ΔM from Stage ① and $\Delta tGlobal$ from ②. This loop can be easily executed in parallel because there are no data dependencies among loop iterations, thus the execution order will not affect the result.

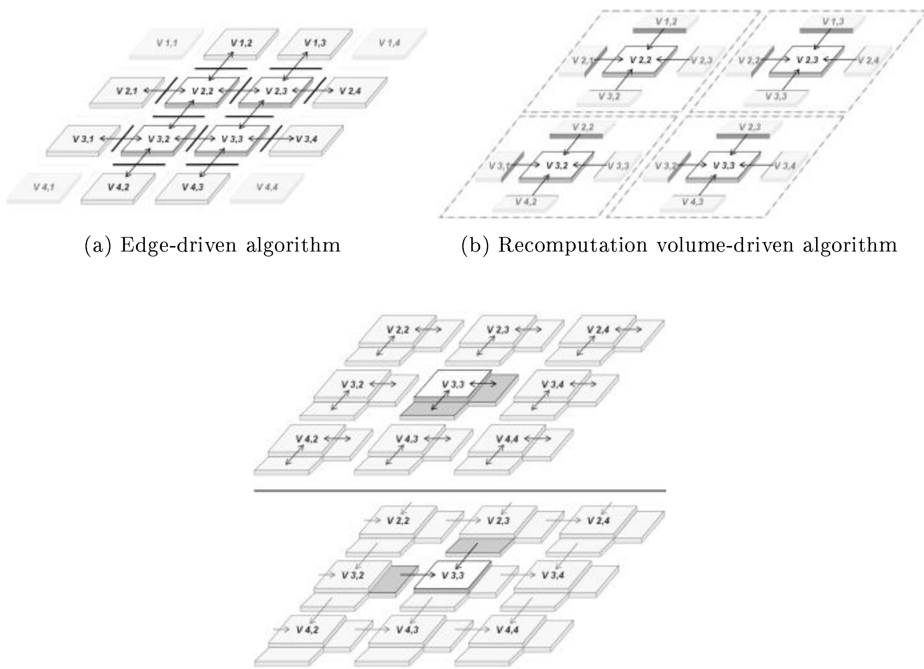
The parallel shallow water simulators presented in this paper were developed in two phases. The first phase consisted in rewriting the edge-driven algorithm into a stream programming model. This phase was described above, the resulting streamed algorithm being depicted in Figure 3(b). The second phase consisted in fine-tuning the streamed code for each particular architecture, a multi-core *CPU* using *OpenMP* (see Section 5.3), a *GPU* using *Brook+* (Section 5.4), and a *GPU* using *OpenCL* (Section 5.5). As we will see, in this process a version based in a privatization approach, depicted in Figure 3(c), was developed.

5.3 Mapping the streamed code on a multi-core CPU using OpenMP

OpenMP is a standard parallel programming extension for shared memory multiprocessor architectures. In *OpenMP* the programmer uses a set of preprocessor directives to instruct the compiler how to generate the parallel code. The main advantage of *OpenMP* is its simplicity, as with little effort it is possible to develop a parallel version of the code.

The streamed code of Figure 3(b) hinges on the recomputation-based solution described in Section 5.2. However, as the *CPU* peak computational power is small when compared to the bandwidth of the memory hierarchy, the streamed algorithm becomes compute bound on the *CPU*. In order to increase the performance, the recomputation strategy was fine-tuned for the *CPU* by applying the privatization-based solution we describe in the next paragraph. The resulting algorithm, shown in Figure 3(c), performs about 30% faster than the recomputation approach for the largest mesh size 4000×2666.

The volume-driven loop on Stage ① in Figure 3(b) was split into two stages. In the first stage (see Stage ①ⓐ in Figure 3(c)), each thread computes the contribution of the right and down edges of the volume v, and stores these contributions in two communication buffers $\Delta MR[v]$ and $\Delta MD[v]$, respectively. In addition, the partial result $\Delta MC[v]$ is computed as $\Delta MR[v] + \Delta MD[v]$. In the second stage (see Stage ①ⓑ in Figure 3(c)), each thread computes the final result $\Delta M[v]$ using its own partial result $\Delta MC[v]$ and the values stored in the two communication buffers $\Delta MR[v]$ and $\Delta MD[v]$. The same applies to the computation of Δt. Figure 4 represents the transformation of the algorithm from the point of view of the finite volume numerical scheme. Figure 4(a) illustrates the initial edge-driven version (see Figure 3(a)), where each edge of the 2D mesh contributes to the solution of its two neighbor volumes. Figure 4(b)



(a) Edge-driven algorithm

(b) Recomputation volume-driven algorithm

(c) Privatization volume-driven algorithm

Figure 4: Parallelization strategies evaluated in this work.

illustrates the behavior of the streamed version (see Figure 3(b)), where each volume of the 2D mesh is processed computing the contribution from its four edges. Figure 4(c) shows the privatization volume-driven algorithm used in the *CPU* (see Figure 3(c)). The algorithm is divided into two stages that use storage buffers to avoid recomputation.

The *OpenMP 2.0* specification for the *C* language does not support the minimum reduction operation used in Stage ②. However, this parallel reduction can be executed efficiently by applying the privatization-based solution (see Section 5.2). First, each thread computes the reduction of a subset of iterations in a private variable. Then, the partial results are combined into one element using a critical section.

5.4 Mapping the streamed code on a GPU using Brook+

Brook+ is a *C* language extension for *AMD GPUs* that exposes a stream programming model. In this paradigm the same function (called the *streaming* *kernel*) is applied to a set of inputs (*input streams*) in parallel, producing another set of outputs (*output streams*). In particular, a thread is created for each output element. The streaming kernel is allowed to read several locations of the input streams but it can only write to one location of each output stream. Thus, the programmer is responsible for writing streaming kernels that are free of race conditions. *Brook+* uses texture memory in order to access input data through *GPU* texture units, a dedicated hardware which provides cached memory access, good *2D* locality or memory access clamping. Although *Brook+* latest version (*v1.4*) permits the utilization of shared memory, it is a beta feature and in our tests it resulted in poor performance or even incorrect results.

The parallelization of Stage ① of the streamed code of Figure 3(b) is as follows. The recomputation-based solution is implemented in *Brook+* by enclosing the loop body in a stream kernel that produces two output streams ΔM and Δt. Each stream kernel invocation processes one volume of the mesh. For this purpose, the four neighbors volumes (up, down, left and right) are fetched (see Figure 4(b) for illustration purposes). The volume mesh can be easily mapped to a 2D stream, which is useful because the described memory access pattern is a kind of stencil operation which has good 2D locality, and this access pattern is very optimized on the texture memory cached access. Furthermore, access clamping is useful to prevent out of range memory access while preserving code regularity (negative coordinates are set to zero and out of range coordinates are set to the maximum allowed value). The resulting code requires fewer conditional statements to process domain boundaries in the *GPU*, reducing branch divergence and avoiding the use of ghost cells in the domain boundaries. The cost of the redundant operations in the recomputation may seem high, but *GPUs* usually have such a large computing power that, even following this approach, some of the resources may remain occasionally unused.

Note that the synchronization-based solution cannot be applied to Stage ① because shared memory is not available, so inter-thread communication would require separate kernel calls and rely on slow global memory. The privatization-based solution is applicable to the *GPU* (see Figure 4(c)), but results about 13% slower than the recomputation-based solution. The privatization approach was previously used in other works (M. de la Asunción et al., 2010). Its main drawback is that the two communication buffers require additional memory and bandwidth. Furthermore, it requires invoking two stream kernels, one for each substage. The cost of calling two *GPU* kernels instead of one is quite high, so the final performance is worse than the recomputation-based solution used in our parallel shallow water simulator. Although the recomputation approach intuitively requires twice as many operations, thanks to expression simplification and *VLIW* (*Very Long Instruction Word*) instruction packing, recomputation only generates about 23% more work than the privatization-based approach, while being about 44% faster thanks to the usage of a single *GPU* kernel. The *VLIW* design of the *GPU* helps to reduce the impact of the additional operations by properly filling the available instruction slots.

The other stages can be easily implemented in *Brook+*. Stage ② performs a reduction operation, which is natively supported by the language. Stage ③ consists of a conflict-free loop that is mapped to the *GPU* by implementing a stream kernel whose code corresponds to the body of the loop (see Stage ③ in Figure 3(b)).

Analogously to the *CPU SSE SIMD* vector instructions, the *GPU VLIW* architecture supports instructions and registers to process several operations in parallel. In addition, it also provides several highly optimized intrinsic operations like scalar product or vector product, which are useful for matrix multiplication and matrix inversion. Usually, *AMD's* compiler does a good job in code vectorization, but after applying the well-known code transformations such as loop-invariant elimination, loop unrolling, common subexpression elimination and symbolic algebra manipulation better *VLIW* slot utilization was achieved. In order to optimize certain vector and matrix operations, it was necessary to write several intrinsic operations explicitly. For example, the four component matrix-vector product (which normally requires 16 products and 12 sums) was rewritten as a set of four scalar products using the *dot* intrinsic.

Finally, a special feature of *GPUs* is that there is an upper bound that limits the number of simultaneous threads, which depends on the hardware and the number of registers used by the kernel. If a kernel uses too many registers, the number of hardware threads decreases. It is possible to fine-tune the kernel code, constrained by the number of registers, to improve the resource utilization. Thus, we have rewritten the code of the streaming kernels to minimize register usage by applying standard compiler transformations which provided about a 10% performance increase. For instance, the Stage ① of the algorithm of Figure 3(b) is the most complex kernel and requires 27 registers, therefore at most 9 wavefronts will be simultaneously executed per *SIMD* processor.

In the final *Brook+* implementation all the stages of the algorithm are executed on the *GPU* and the *CPU* is only used to maintain the state of the simulation. Such state is used to configure the launch of the *GPU* kernels, to write the simulation data to disk at regular intervals, and to display some status information about the process. Thus, few device memory transfers are needed.

An interesting feature of the *GPU* implementation is the use of an additional thread to perform disk writes. Depending on the desired time intervals to write the state of the simulation to disk, many frequent disk writes could slow down the execution if the computation has to wait until the dump of the mesh state is complete. Our approach consists in dumping the information in an output buffer and using a thread to handle the write operations asynchronously, so that the execution can continue.

5.5 Mapping the streamed code on a GPU using OpenCL

OpenCL (Khr, 2011) is a widely supported standard that enables the execution of *C* parallel code on different heterogeneous systems with minimal effort. For optimal performance, it is highly recommended to tune the *OpenCL* code for the hardware platform. Thus, an *OpenCL* implementation that executes efficiently on a *CPU* may not offer good performance on a *GPU* due to, for example, thread divergence, memory coalescence issues or shared memory bank conflicts. The *Brook+* language provides a streaming model that eases *GPU* programming. In contrast, *OpenCL* provides more flexibility and control to the programmer by exposing some *GPU*-specific hardware features. For instance, *Brook+* always stores data in texture memory, so texture cache is enabled by default. In contrast, the *OpenCL* programmer has to specify which data will be stored in texture memory. Similarly, *OpenCL* enables one to manipulate the *GPU* shared memory and to use intra-block synchronization primitives to avoid race-conditions. These *OpenCL* features enable the implementation of fast data communications, which can significantly impact on performance.

Hereafter, two *OpenCL* implementations based on the privatization and recomputation strategies are described. As in this work the same *GPU* will be used to test our *Brook+* and *OpenCL* implementations, both versions share some low level optimizations, such as the *VLIW* friendly code and the register reduction. Specifically, the *OpenCL* version includes the standard software changes to configure the runtime, manage the memory buffers, perform the online code compilation and launch the kernels.

The privatization strategy was fine-tuned for *OpenCL* in order to improve performance, by taking advantage of the shared memory for fast intra-block communications and for reducing global memory bandwidth utilization. Observe in Figure 3(c) that the algorithm requires two separate stages (Stage ①ⓐ and Stage ①ⓑ) to compute the partial flow contributions (ΔM) and local time steps (Δt) for each volume. The *Brook+* implementation relies on slower global memory, requiring a second kernel to integrate the results stored in the communication buffers. However, in *OpenCL*, these communication buffers can be stored in shared memory, which provides a fast and efficient way to obtain the partial volume contributions in a single kernel.

The amount of shared memory for each *OpenCL* block of threads is very limited, therefore the domain must be divided in smaller rectangular subdomains which are distributed among the blocks of threads. Intra-block communication will be performed in shared memory. However, inter-block communications are not supported by the language. This is solved by performing some recomputation for the cells that reside in the edge of each block, which makes unnecessary the communications between blocks. As illustrated in Figure 5, an additional left column and upper row of ghost cells is added to each block (see for example cells v3.3, v3.4, v3.5, v4.3 and v5.3 in Figure 5). These cells define a replicated region which is common to each two adjacent blocks and provides the required flow and pollutant contribution without involving any communication between the blocks. The solution shares some similarities with the recomputation strategy, which also eliminates the need for inter-thread communication. However in this case, to minimize the number of redundant operations, the recomputation is restricted to the edges where the block of threads is expecting a flow contribution from the neighbor block. A block size of 8×8 threads will be used in our *GPU* (equal to the hardware wavefront), which enables to remove the intra-block synchronizations primitives between the kernel of Stage ①ⓐ and Stage ①ⓑ. Notice that due to the additional ghost cells, only 49 out of 64 threads from each block perform useful work (that is, a 7×7 region). This fact causes some thread divergence and reduces a bit the execution speed.

In addition to the previous tweak, the Stages ①ⓐ and ①ⓑ have been fine-tuned to reduce the memory bandwidth consumption as follows. Each thread block of Stage ①ⓐ now writes a single Δt value that summarizes the minimum of all the thread-private Δt values. As a result, the workload of the final global time step $\Delta tGlobal$ reduction of Stage ② is decreased accordingly. Regarding this reduction of Stage ②, while *Brook+* has some built-in support for reductions, in *OpenCL* the programmer has to implement his own reductions kernels. In this work, an efficient parallel reduction implementation which takes advantage of shared memory and minimizes synchronizations was used (multi-stage reduction algorithms are very common in *OpenCL* and *CUDA*). Finally, Stage ③ updates the volume mesh with $\Delta tGlobal$ and does not present any relevant modification.

The recomputation-based strategy does not take advantage of the *GPU* shared memory. Therefore, no significant changes other than the reduction

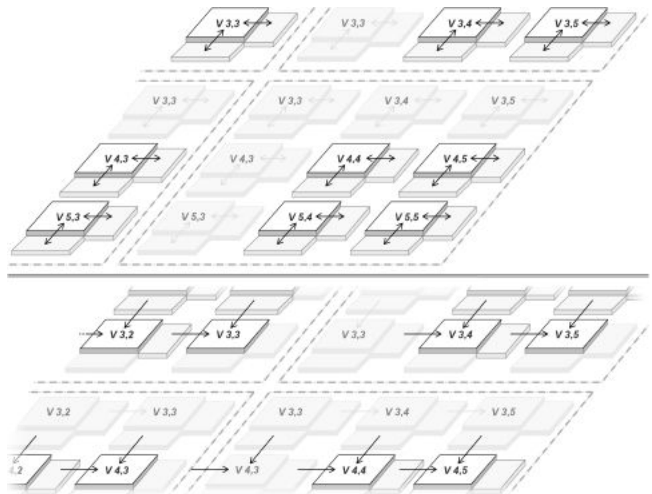


Figure 5: Optimized privatization volume-driven algorithm with block border replication.

of Stage ② were made. For most mesh sizes the recomputation strategy results around 2.5% faster than the privatization strategy described above, mostly due to the better *VLIW* instruction packing.

6 Experimental results

Our test platform is composed of a *Core i7 950* quad-core *CPU* running at 3.06 *GHz*, with 6 *GB DDR3 1866 CL9* memory, a *X58* chipset based motherboard and a *Radeon 5870 GPU*. The *Radeon 5870* is an *AMD GPU* with 1600 processing elements (distributed in 20 *SIMD* processors, each one having 16 cores with *5-way VLIW* support). The software setup is *Windows XP x64* operating system, using *Microsoft Visual C++ 2008* compiler (x64, release profile), *Brook+ 1.4* and *AMD's OpenCL SDK 2.5* with the *Catalyst 11.11 GPU* driver.

The *CPU/OpenMP* parallel implementations (Section 5.3) are built with *OpenMP* directives and *SSE*-based *SIMD* instructions inserted by the automatic vectorization capabilities of the compiler. It is run on 8 threads on the *Core i7* processor with hyper-threading (4 cores × 2 threads per core). Hyper-threading enables the parallel execution of two threads per core, although typically the second thread only provides between 5% and 20% the performance of a real core. As will be shown later in Section 6.1.1, there are no significant numerical differences between running our application with single or double pre-

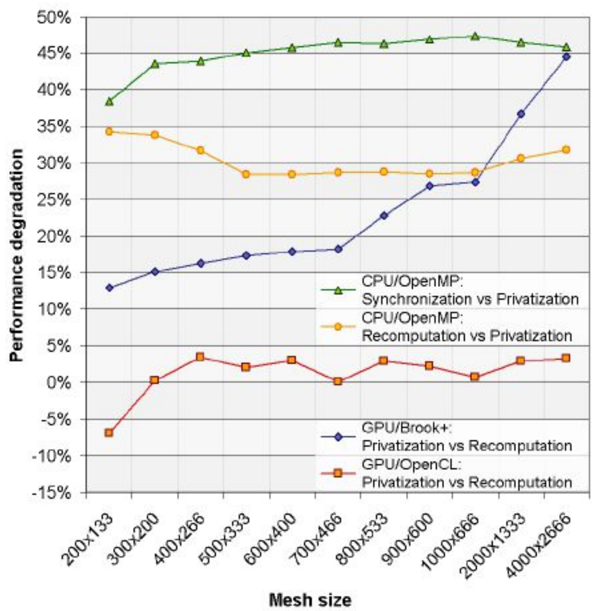


Figure 6: Pairwise comparison of the different parallelization strategies for each platform.

cision, while double precision has a large negative impact on GPU performance. According to our experiments, some double precision operations like divisions cannot be directly handled by the GPU hardware and will generate a sequence of instructions to compute the result, thus heavily reducing performance. In fact we have measured around 35 times larger runtimes when our application is run in the GPU using double precision. Therefore, the benchmark results are presented for single precision. The *GPU/Brook+* (Section 5.4) and *GPU/OpenCL* (Section 5.5) parallel implementations are run on the *Radeon 5870* with *VLIW* code generation enabled. The execution time includes all data transfers between *CPU* and *GPU* memory. Nonetheless, to prevent benchmark contamination, the evolution of the simulation is not written to disk, otherwise the result would be largely dependent on disk performance, specially for small problems.

Several parallelization strategies were described in Section 5 for each platform. Thereafter, the best strategy must be selected for *CPU* using *OpenMP*, for *GPU* using *Brook+* and for *GPU* with *OpenCL*. Figure 6 presents a pairwise comparison for several mesh sizes, showing the relative performance degradation of one strategy compared to another. For example, a positive performance degradation for *CPU/OpenMP* Synchronization vs Privatization means that synchronization is slower than privatization. For the *CPU*, the performance degradation of the synchronization and recomputation strategies compared to privatization is presented. The privatization solution is clearly the fastest, followed by recomputation which is about 30% slower and synchronization which is about 45% slower. Regarding *GPU/Brook+*, language restrictions made it impossible to implement a synchronization strategy in the *GPU*. Thus, the figure compares privatization against recomputation. Observe that in this case recomputation is the fastest, and that privatization is between 15% and 45% slower. Finally, for *GPU/OpenCL* the performance of recomputation and the optimized privatization algorithm is quite similar. Nonetheless, recomputation is up to 3% faster for mesh sizes larger than 300×200. The reason why recomputation is faster than privatization is that due to the impressive arithmetic power of the *GPU*, it is cheaper to take advantage of the *Radeon VLIW* execution to process all the data.

The rest of this section evaluates the performance of our parallel shallow water simulator in terms of execution time and speedups. The speedups are computed with respect to the best sequential implementation of the shallow water simulation (*CPU/Seq*), which is the sequential execution of the privatization volume-driven algorithm of Figure 3(c). According to our experiments, *CPU/Seq* is faster than the original sequential version of the numerical algorithm. Hereafter, the notation *CPU/OpenMP* will refer to the privatization, *GPU/Brook+* to the recomputation algorithm, and *GPU/OpenCL* to the recomputation algorithm, all of which are volume-driven.

6.1 Academic 2D problem: Simulator verification

In this section a simple test case is presented to verify the accuracy of the simulator. The test consists of a dam-break problem where a water column falls in a water tank creating a series of ripples that can be easily examined. We use a small $[-5,-5] \times [5,5]$ domain with the depth function defined as:

$$H(x,y) = 1 - 0.4e^{-x^2-y^2}, \tag{36}$$

and with the following initial condition:

$$W(x,y,0) = \begin{pmatrix} h(x,y,0) \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{37}$$

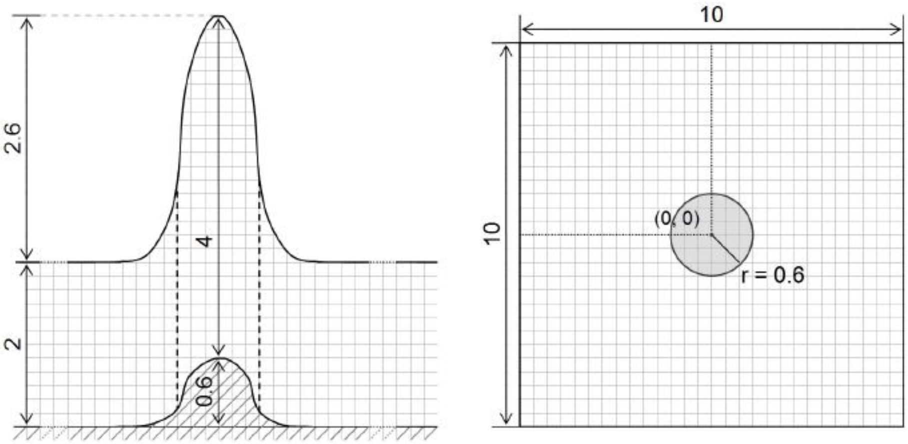


Figure 7: Diagram of the academic 2D problem used for verification.

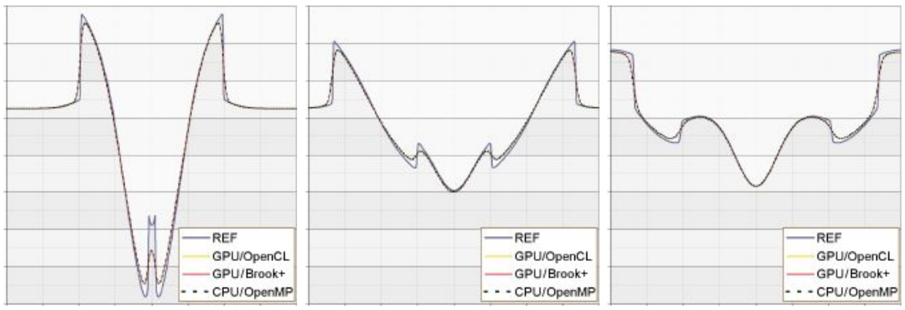
where

$$h(x,y,0) = \begin{cases} 4 & \text{when } x^2 + y^2 \leq 0.36 \\ 2 & \text{otherwise} \end{cases} . \tag{38}$$

and the size of the side $\Delta x = \Delta y = 10$ / *number of volumes per side.*

6.1.1 Numerical results

The simulations are executed in the time interval $[0, 1]$ for several mesh sizes using wall boundary conditions ($\boldsymbol{q} \cdot \boldsymbol{\eta} = 0$) and $CFL = 0.9$. Figure 7 shows a diagram of the initial setup. This test does not require wet-dry zone processing and serves to study the proper behavior of the forces and conservation of the fluid. Figure 8 represents the evolution of the test showing a bisection plane of the domain for 0.33, 0.66 and 1.00 seconds. In each figure there are three lines representing the water height: the *REF* version (thick solid line), which is *CPU/Seq* using a very fine mesh of 3200×3200 volumes and the initial waves were purposely quite high and sharp to be able to observe the behavior in the test; *CPU/OpenMP* (dashed line) using a 400×400 mesh size; *GPU/OpenCL* (thin light line) using a 400×400 mesh size; and *GPU/Brook+* (thin dark line) using a 400×400 mesh size. The *CPU/OpenMP*, *GPU/OpenCL*, and *GPU/Brook+* simulations are equivalent as their lines are always overlapped. However, both of them present a slight difference with respect to the reference solution, specially in the inflection points, where their contour tends to be more rounded.



(a) Time = 0.33 seconds (b) Time = 0.66 seconds (c) Time = 1.00 seconds

Figure 8: Evolution of the academic 2D problem used for verification.

Table 1: L^1 error at time $T = 1$ for mesh 400×400 in *CPU/OpenMP* and *GPU/Brook+* using single and double precision. The reference solution is *CPU/OpenMP* in double precision using mesh 3200×3200.

L^1 error	*GPU/Brook+* single	*CPU/OpenMP* single	*CPU/OpenMP* double
h	1.4067375E-02	1.4067372E-02	1.4067382E-02
q_x	1.7777456E-02	1.7777453E-02	1.7777532E-02
q_y	1.7777455E-02	1.7777509E-02	1.7777532E-02

On the other hand, the numerical error due to the use of single and double precision is analyzed using two embedded meshes: a very fine reference mesh of 3200×3200 volumes, and a coarser mesh of 400×400 volumes. The reference solution for the precision analysis is *CPU/OpenMP* in double precision for mesh size 3200×3200. Table 1 details the L^1 norm error for variables h, q_x, q_y and mesh size 400×400. The error is computed for *GPU/Brook+* in single, *CPU/OpenMP* in single and *CPU/OpenMP* in double, with respect to the reference solution (*GPU/OpenCL* is not presented here as the results were very similar to the *GPU/Brook+* version). It can be observed that in the three cases the error has the same order and there is no significant difference between the *CPU* and the *GPU* or between single and double precision.

Table 2 details the L^1 norm error between *GPU/Brook+* in single precision and *CPU/OpenMP* in double precision when both are using the same mesh size. In this case it can be observed that the error has the order of the single precision limit, so for our finite volume simulations, it is enough to compute using single precision. This is due to the fact that almost all operations performed by the

Table 2: L^1 error at time $T = 1$ for meshes 400×400 and 3200×3200 in *GPU/Brook+* with single precision, being *CPU/OpenMP* in double precision the reference solution.

L^1 error	400×400	3200×3200
h	2.1429375E-07	5.4036922E-06
q_x	3.0220140E-07	4.6754546E-06
q_y	3.1466585E-07	4.7149664E-06

algorithm are basic operations (like additions and multiplications), and there are few *GPU* transcendental functions involved (whose precision may be lower). The *GPU* arithmetic conforms with the *IEEE-754* standard except for a little rounding deviation in some intrinsic functions.

6.1.2 Performance results

Table 3 shows the total execution time (expressed in seconds) of *CPU/Seq*, *CPU/OpenMP*, *GPU/Brook+* and *GPU/OpenCL* for several mesh sizes. The speedups are computed with respect to the best sequential implementation *CPU/Seq*. The *Num. Iter.* column indicates the number of iterations performed by the algorithm to complete the simulation. Observe that smaller tests are specially efficient on the *GPU/OpenCL* implementation, which is only outperformed by *GPU/Brook+* above the 1600×1600 mesh.

The first mesh size to take longer than one second on the *GPU* is 600×600 for *GPU/Brook+* and 700×700 for *GPU/OpenCL*. However, these simulations already require more than 40 seconds on *CPU/OpenMP*. For the largest simulation, *GPU/Brook+* finishes in ≈ 77 seconds and *GPU/OpenCL* in ≈ 97 seconds, while *CPU/OpenMP* requires almost two hours. *CPU/OpenMP* tends to obtain speedups slightly greater than 4x on the quad-core *CPU* thanks to the use of hyper-threading. *GPU* speedups are huge (up to 388x for *GPU/Brook+* and 308x for *GPU/OpenCL*), although we must remember that this is only a short test simulation which uses neither wet-dry fronts nor pollutant transport.

Notice how the *GPU* speedups increase quickly from one mesh size to another. Thereby, in order to obtain good processor utilization in the *GPU*, we should work with large domains that create at least about hundreds of thousands threads. *GPUs* can execute a large number of threads efficiently because they rely on latency hiding techniques like interleaved multi-threading, which switch among threads in order to perform useful work during the time required to complete dependent arithmetic operations and memory requests.

Table 3: Academic 2D dam-break problem: Execution times (in seconds) and speedups for the *CPU/OpenMP* and *GPU/Brook+* implementations.

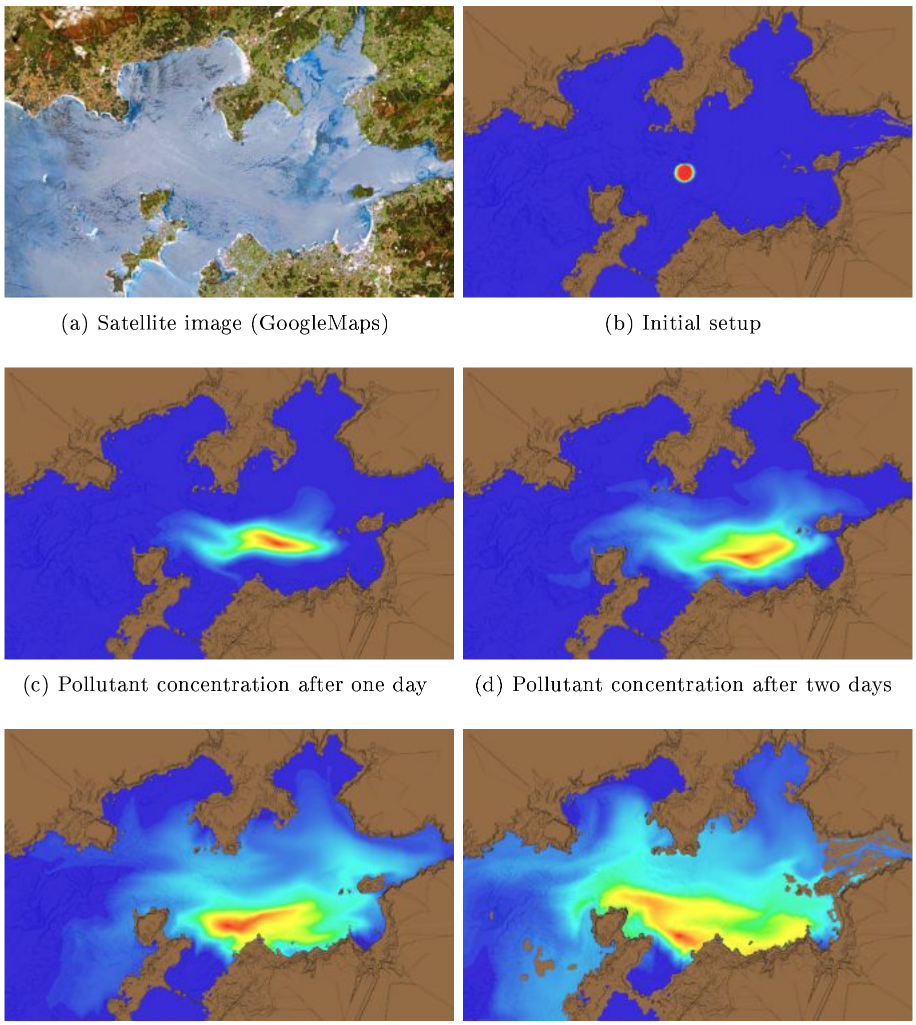
Mesh size	Num. Iter.	*CPU/Seq* time	*CPU/OpenMP* time/speedup		*GPU/Brook+* time/speedup		*GPU/OpenCL* time/speedup	
200×200	246	6.8	1.6	4.2	0.4	15.5	0.1	113.3
300×300	372	23.2	5.3	4.4	0.6	36.2	0.1	210.9
400×400	498	55.0	12.6	4.4	0.8	70.5	0.2	250.0
500×500	623	107.1	23.9	4.5	1.0	112.7	0.4	255.0
600×600	749	185.7	41.4	4.5	1.2	151.0	0.7	269.1
700×700	875	294.8	64.6	4.6	1.6	190.2	1.1	270.5
800×800	1001	439.9	97.0	4.5	2.1	213.6	1.6	283.8
900×900	1127	626.3	138.3	4.5	2.6	238.1	2.2	285.9
1000×1000	1253	859.1	190.4	4.5	3.3	259.5	3.1	273.6
1600×1600	2009	3522.7	770.7	4.6	10.8	325.9	12.6	278.7
2000×2000	2514	6873.1	1516.7	4.5	19.8	347.7	24.4	281.2
2700×2700	3396	16287.8	3587.6	4.5	46.5	350.1	58.6	277.9
3200×3200	4027	29887.4	6626.9	4.5	76.9	388.5	97.1	307.8

6.2 Synthetic problem: Ría de Arousa in Galicia (Spain)

The second test uses a synthetic case in order to study the efficiency in a real world scenario. The simulation is based on an actual estuary in Northwest Spain called the *Ría de Arousa*, whose satellite image is displayed in Figure 9(a). This natural environment is simulated using the real terrain and bathymetry data in our test. While the north and east limits of the area involved in the simulation have free boundary conditions, the tides in the west and south borders are simulated using the main barotropic tidal components. Wet-dry fronts appear very often in this test in the coastal zones and emerging islands. The purpose of the simulation is to study the evolution of a pollutant that is discharged in this environment, determining its propagation and which are the most affected ares. The total simulated period is one week of real time.

6.2.1 Numerical results

The initial setup is represented in Figure 9(b). It corresponds to the moment when the pollutant is discharged in a circle with a radius of 400 m in the middle of the estuary. The normalized concentration of pollutant is given by the color scale at the bottom of the figure. Figure 9(c) is a capture of our simulation after 24 hours. Here the sea currents have started to extend the pollutant along the estuary, but if containment measures and cleanup activities started at this moment, it would be possible to safely remove a large part of the contaminant.



(a) Satellite image (GoogleMaps)

(b) Initial setup

(c) Pollutant concentration after one day

(d) Pollutant concentration after two days

(e) Pollutant concentration after four days

(f) Pollutant concentration after eight days

Figure 9: Evolution of the Ría de Arousa simulation.

After another 24 hours of simulated time we reach the situation depicted in Figure 9(d), where the pollutant has spread further, increasing portions of it beginning to reach the seashore. Cleaning efforts could still be concentrated in a well defined zone and remove most of the contamination. In Figure 9(e), four days after the spill, the pollutant has spread over a large area, but a reasonable amount of waste material could still be drawn from the center of the stain. Cleaning activities can begin in some coastal zones too. After eight days (see Figure 9(f)) the damage is extensive and only a few areas remain relatively safe, such as the two north bays and the south one. Now most of the shore requires cleaning efforts, specially the south zone, but depending on the toxicity of the pollutant the process may have reached catastrophic dimensions. The test benchmark only simulates seven days, but here we used the eighth day to display an image during low tide, in which we can observe some small emerging islets. The model has provided a valuable simulation of the disaster evolution that makes possible to predict the most affected areas. Pollutant discharge may not only have a deep impact on the natural environmental, but also affect very negatively the economy of regions where seafood products or tourism are relevant industries.

6.2.2 Performance results

Table 4 shows the execution times and speedups of several mesh sizes. The numbers in *Num. Iter.* show that this second problem requires about 1000 times more iterations. Although the *GPU* speedups on big meshes are lower than the ones observed in the academic problem, they are excellent for a realistic test case. *CPU/OpenMP* also offers good speedups, more than 4x for 4 cores, but it only results adequate for very small simulations. The smallest mesh (200×133) takes just 59 seconds for *GPU/OpenCL*, around 158 seconds for *GPU/Brook+* and more than 16 minutes for *CPU/OpenMP*. For 4000×2666, a simulation that takes about a year with *CPU/Seq* would take nearly 3 months using *CPU/OpenMP*, but it would be reduced to only 39.5 hours with *GPU/Brook+* or 43.7 hours with *GPU/OpenCL*. Note that both *GPU* implementations enable real-time shallow water simulations, as for a 7-day period these simulations require less than 2 days. These impressive results make it possible to perform complex simulations over long periods of time within reasonable execution times.

Figure 10 compares the performance results of *CPU/Seq*, *CPU/OpenMP*, *GPU/Brook+* and *GPU/OpenCL* in terms of execution time for several mesh sizes using a logarithmic scale. Notice that the gap between *CPU* and *GPU*

Table 4: Synthetic problem: Execution times (in seconds) and speedups for the *CPU / OpenMP*, *GPU / Brook+* and *GPU / OpenCL* implementations.

Mesh size	Num. Iter.	*CPU / Seq* time	*CPU / OpenMP* time/speedup		*GPU / Brook+* time/speedup		*GPU / OpenCL* time/speedup	
200×133	335514	4477	978	4.6	158	28.4	59	75.3
300×200	503362	14665	3159	4.6	270	54.4	121	121.3
400×266	671293	34752	7657	4.5	457	76.1	229	151.8
500×333	839236	68684	14805	4.6	691	99.4	409	168.0
600×400	1007255	117995	25839	4.6	954	123.7	643	183.5
700×466	1175349	186205	40730	4.6	1338	139.2	1008	184.8
800×533	1343455	277485	60570	4.6	1867	148.7	1412	196.5
900×600	1511582	395342	85776	4.6	2346	168.5	1986	199.0
1000×666	1679709	541479	117730	4.6	3160	171.4	2804	193.1
2000×1333	3361568	4305369	940258	4.6	19799	217.5	21157	203.5
4000×2666	6727438	33895851	7405598	4.6	142246	238.3	157289	215.5

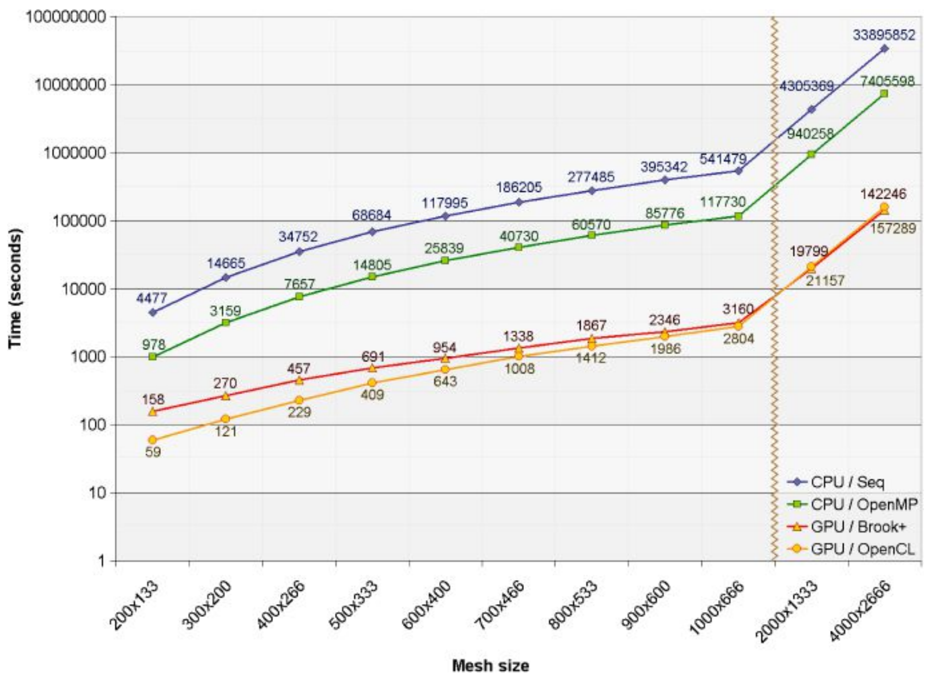


Figure 10: Execution time (in seconds) for several mesh sizes.

widens as the number of finite volumes grows (specially for *GPU / Brook+*) because *GPUs* offer better performance when working on big domains, where they are able to make a better use of their execution resources and latency hiding techniques. For mesh sizes up to 1000×666 *GPU/OpenCL* outperforms *GPU/Brook+*, *GPU/Brook+* being slightly ahead for the finest meshes 2000×1333 and 4000×2666. The reasons for this behavior have to do with restrictions of the *Brook+* programming language, namely, the use of the graphics pipeline (in particular, kernel invocation overhead, memory tiling and thread blocking) to process data and the restricted control over memory allocation and *CPU-GPU* transfers. Thus, the impact of *CPU-GPU* transfers and kernel invocation on performance is smaller as the problem size grows. Furthermore, the implicit memory tiling (hierarchical-Z pattern) and thread blocking of pixel shaders offers good 2D spatial locality, which leads to slightly better efficiency for larger meses.

6.2.3 Performance scaling and limiting factors

The analysis of the performance scaling depending on the amount of execution resources and the application profiling can be used to estimate the performance limiting factors, which is useful to find out which parts of the implementation may be subject to further optimization.

An interesting performance metric is the number of cells that can be processed per unit of time, which in our case will be expressed in *Mcell/s* (million cells per second). The processing rate of *CPU/Seq* remains quite stable at about 2.05 *Mcell/s*, this points to some kind of performance limiting factor which leads to a fixed processing rate. The *CPU/OpenMP* version enables to use additional computational resources, but the required memory bandwidth is also increased according to the amount of threads. The cell processing rate of *CPU/OpenMP* using 8 threads is also quite stable at around 9.4 *Mcell/s*, which results in about a 4.6x speedup. If hyperthreading is disabled to obtain a more accurate scaling factor depending only on the number of real cores, the thread processing rate drops to about 7.75 *Mcell/s*. The resulting application speedup is nearly 3.8x, and remains fairly stable as the mesh size increases. The good performance scaling of *CPU/OpenMP* (95% of a perfect speedup) points to the computing power as the main limiting factor. To confirm this, further tests were conducted to find out the cache miss ratio of the last cache level in the larger meshes. Thanks to the predictability of the memory access pattern and the *CPU* data prefetchers, the *L*3 miss ratio was only about 1% of the memory requests.

On both *GPU/Brook+* and *GPU/OpenCL* the cell processing rate does not seem to stabilize and keeps steadily growing as the mesh size increases. For the smallest mesh size *GPU/OpenCL* obtains only 151 *Mcell/s*, while for the largest mesh it achieves 456 *Mcell/s* and the *GPU/Brook+* version exceeds 500

Mcell/s. The lower initial results can be attributed to some fixed execution costs, such as *GPU* initialization, kernel launch times and memory transfers. *GPU* architectures largely rely on their multi-threading capabilities to improve efficiency and hide memory latency, therefore it is normal to obtain better resource utilization for the larger meshes. Notice that the *GPU* speedups are not directly comparable to the *CPU* architecture, where performance scaling was directly related to the number of cores. Although the *GPU* has many times more processing resources, the architecture was designed for parallel execution, with simpler and slower cores that make quite difficult to achieve nearly perfect scaling or peak performance in real-world applications. In fact, the cores of the *Radeon 5870 GPU* work at 850 *MHz*, a much lower clock frequency compared to the 3.06 *GHz* of the *Core i7 950 CPU*. These *GPU* cores largely rely on *SIMD* execution, therefore runtime code divergence is processed sequentially, leading to a significant performance penalty when complex control flow is involved. Moreover, the *VLIW* design of the architecture makes very difficult to achieve high resource utilization in real applications. In our tests, the recomputation strategy achieves an average of 3.97 slots per 5-way *VLIW* instruction, which is a very good utilization rate for the architecture.

The *CPU/OpenMP* implementation seems to be clearly compute-bound, however, it is not so evident to determine whether *GPU* solutions are compute-bound or memory-bound. Although the *GPU* memory bandwidth utilization is higher than the *GPU* computing resource utilization, *GPUs* usually can cope well with high bandwidth utilization. To provide more information about the limiting factor in *GPU/Brook+*, Figure 11 shows, for all mesh sizes, the performance degradation experienced when lowering the *GPU* core clock by 15%, the *GPU* memory clock by 15%, or both clocks at the same time for several mesh sizes. As can be observed, the mesh size rapidly increases the impact of the core clock reduction, while the influence of the memory clock reduction is smaller. Furthermore, the impact of core clock reduction (11% in the worst case) is nearly twice the impact of memory clock reduction (about 6% in the worst case). This fact suggests that the performance of the *GPU* is more bound by computing power than by the memory bandwidth. Finally, note that a 15% reduction in both core and memory clocks results in a 17% performance drop. This information provides an estimation of the architecture scalability with respect to the clock speed in our application. The opposite should also be true, and increasing both parameters by 15% instead of reducing them, will probably result in about a 15% performance improvement, which is a very good scaling.

Further improvements in the parallel implementations should focus on the

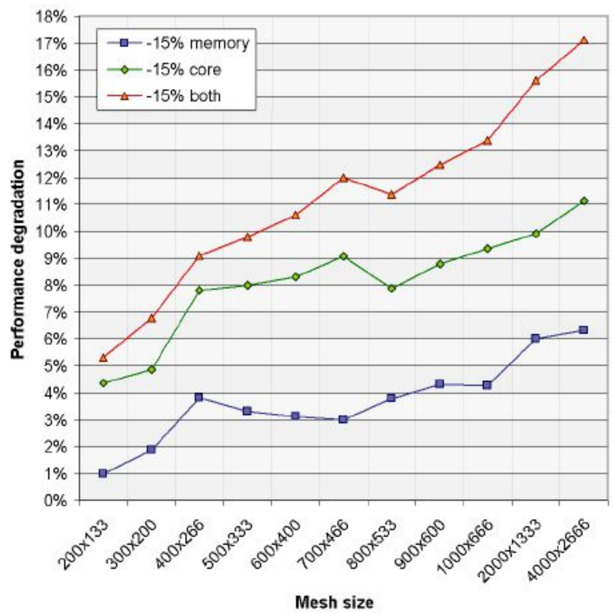
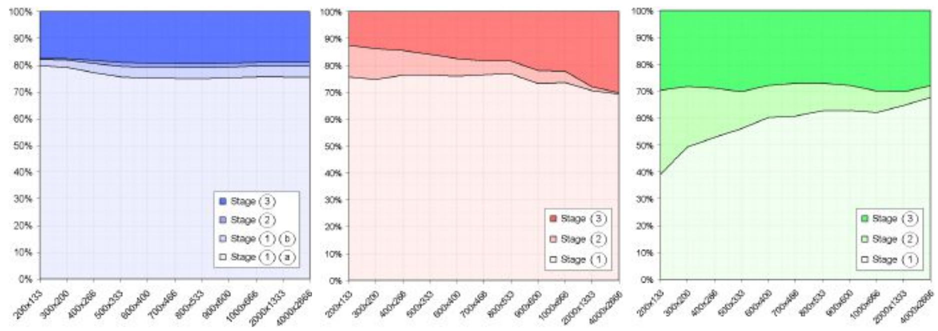


Figure 11: Performance impact of the GPU core and memory clock frequency on performance.

most costly parts of the algorithm. Figure 12(a) shows the relative computational cost for each stage of our *CPU/OpenMP* parallel implementation, Figure 12(b) for *GPU/Brook+* and Figure 12(c) for *GPU/OpenCL*. The three figures present the information for all mesh sizes, thus, it is possible to study performance factors that depend on the problem size. It can be observed that in *CPU/OpenMP* the relative cost of the stages is independent of the mesh size. Thus, the cost of *Stage* ①ⓑ is very low (about 5%) and the cost of *Stage* ② is negligible (about 1%). Most work is done by *Stage* ①ⓐ (about 75%), the remaining 19% being consumed by volume update of *Stage* ③. In the case of *GPU/Brook+*, the relative cost is more dependent on the problem size. *Stage* ① consumes between 70% and 76% of the total time, even though we are using the recomputation strategy. The cost of the reduction operation of *Stage* ② decreases as the mesh size grows, initially consuming nearly 15% of the execution time, but being negligible for the biggest mesh size 4000×2666. For *GPU/OpenCL*, the relative cost of each kernel is even more dependent on the problem size. In particular, the reduction of *Stage* ② consumes around 35% of the execution time for the smaller meshes, while it consumes only 4% for the largest mesh size. Some kernel optimizations may be still be possible in *Stage*



(a) *CPU/OpenMP* (b) *GPU/Brook+* (c) *GPU/OpenCL*

Figure 12: Relative computational cost of each stage of the algorithm.

②. Nonetheless, let us observe that for the 4000×2666 mesh the distributions of the relative cost of the kernels of *GPU/Brook+* and *GPU/OpenCL* are very similar.

7 Conclusions and future work

This paper describes a numerical scheme of a shallow water simulator that is able to handle wet-dry zones as well as the transport of inert substances such as a pollutant on a river. These simulations have great interest in many industrial and environmental projects, but unfortunately they have a very high computational cost. This motivates our proposal of efficient and cost effective parallel implementations of this scheme in multicore and manycore systems.

The development of the parallel shallow water simulator was driven by an analysis based on domain-independent kernels, which is a useful tool that provides valuable information to the programmer to find conflictive structures and adopt the best parallelization strategy. Several parallelization strategies were described to take advantage of current multi-threaded systems and to make an efficient processing of complex simulations in a fraction of the time required by the sequential algorithm. A speedup of up to 4.6x was obtained using *OpenMP* on four cores with hyper-threading. An impressive 238.3x speedup was obtained using a *GPU* with *Brook+* with respect to the sequential version (52.1x if compared to the *OpenMP CPU* version), and an excellent 215.5x was obtained using the same *GPU* with *OpenCL* (47.1x if compared to the *OpenMP CPU* version).

This work also demonstrates how streaming code developed to be executed in a general purpose processor using platform independent optimizations can be modified to run in *GPUs*, reducing simulation time by more than two orders of magnitude. Using the same programming model and algorithms on both architectures facilitates fast application portability.

There are many interesting topics as future work, such as the use of higher order models to improve simulation accuracy or the modification of the application to support two flow layers in order to enable the simulation of other complex problems, such as oceanic currents.

Acknowledgments

We would like to thank the G-HPC network for promoting interdisciplinary collaborations between groups of the network. This research has been supported by the Galician Government (Consolidation of Competitive Research Groups, Xunta de Galicia ref. 2010/6) under projects INCITE08PXIB105161PR and 08TIC001206PR, the Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union under the grant TIN2010-16735, and the projects MTM2009-11923 and MTM2010-21135. Finally, we also thank the Consellería do Mar of Xunta de Galicia (local government of Galicia, Spain) and the Centro Tecnolóxico do Mar (CETMAR) for providing the ocean currents and topographic data of Ría de Arousa.

References

A. Kurganov and G. Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Commun. Math. Sci.*, 5(1):133–160, 2007.

AMD Stream Computing User Guide. AMD, 2009. v1.4.0a.

A.R. Brodtkorb, M.L. Sætra, and M. Altinakar. Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification and Validation. *Computer and Fluids*, 55:1–12, 2012.

D. Ribbrock, M. Geveler, D. G oddeke, and S. Turek. Performance and Accuracy of Lattice-Boltzmann Kernels on Multi- and Manycore Architectures. *International Conference on Computational Science. Procedia Computer Science*, 1(1):239–247, 2010.

D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke, and C. Gutwenger. HONEI: A Collection of Libraries for Numerical Computations

Targeting Multiple Processor Architectures. *Computer Physics Communications*, 180(12):2534–2543, 2009.

E.F. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows.* John Wiley & Sons, 2001.

J. Setoain, C. Tenllado, J.I. Gómez, M. Arenaz, M. Prieto, and J. Touriño. Towards Automatic Code Generation for GPU Architecture. In *Proc. of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008.

J.M. Gallardo, S. Ortega, M. de la Asunción, and J.M. Mantas. Two-Dimensional Compact Third-Order Polynomial Reconstructions. Solving Nonconservative Hyperbolic Systems Using GPUs. *Journal of Scientific Computing*, pages 1–23, 2011.

The OpenCL Specification, version 1.2. Khronos OpenCL Working Group, 2011.

M. Arenaz, J. Touriño, and R. Doallo. Compiler Support for Parallel Code Generation through Kernel Recognition. In *Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium*, pages 79b (CD–ROM, 10 pages), 2004.

M. Arenaz, J. Touriño, and R. Doallo. XARK: An EXtensible Framework for Automatic Recognition of Computational Kernels. *ACM Transactions on Programming Languages and Systems*, 30(6):1–56, 2008.

M. de la Asunción, J.M. Mantas, and M.J. Castro. Simulation of One-Layer Shallow Water Systems on Multicore and CUDA Architectures. *Journal of Supercomputing*, pages 1–9, 2010.

M. Geveler, D. Ribbrock, D. G oddeke, and S. Turek. Lattice-Boltzmann Simulation of the Shallow-Water Equations with Fluid-Structure Interaction on Multi- and Manycore Processors. *Lecture Notes in Computer Science: Facing the Multicore Challenge*, 6310:92–104, 2010.

M. Lastra, J.M. Mantas, C. Ureña, M.J. Castro, and J.A García-Rodríguez. Simulation of Shallow Water Systems using Graphics Processing Units. *Mathematics and Computers in Simulation*, 80(3):598–618, 2009.

M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, and R. Doallo. Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture. In *Proc. of Workshop on Exploitation of Hardware Accelerators (WEHA 2011)*

As part of the 2011 International Conf. on High Performance Computing and Simulation, HPCS2011, pages 664–670, 2011.

M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, and C. Parés. A Parallel 2D Finite Volume Scheme for Solving Systems of Balance Laws with Nonconservative Products: Application to Shallow Flows. *Computer Methods in Applied Mechanics and Engineering*, 195(19-22):2788–2815, 2006.

M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, and C. Parés. Solving Shallow-Water Systems in 2D Domains using Finite Volume Methods and Multimedia SSE Instructions. *J. Comput. Appl. Math.*, 221(1):16–32, 2008a.

M.J. Castro, T. Chacón, E.D. Fernández-Nieto, J.M. González-Vida, and C. Parés. Well-Balanced Finite Volume Schemes for 2D non-Homogeneous Hyperbolic Systems. Application to the dam break of Aznalcóllar. *Computer Methods in Applied Mechanics and Engineering*, 197:3932–3950, 2008b.

M.J. Castro, E.D. Fernández-Nieto, A.M. Ferreiro, J.A. García-Rodríguez, and C. Parés. High Order Extensions of Roe Schemes for Two Dimensional Nonconservative Hyperbolic Systems. *Journal of Scientific Computing*, 39(1): 67–114, 2009.

M.L. Sætra and A.R. Brodtkorb. Shallow Water Simulations on Multiple GPUs. In *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 56–66. Springer, 2012.

R. Chandra, L. Dagum., D. Kohr., D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers Inc., 2001.

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

T. Morales de Luna, M.J. Castro, C. Parés, and E. Fernández Nieto. On a Shallow Water Model for the Simulation of Turbidity Currents. *Communications in Computational Physics*, 6:848–882, 2009.

T. Runar, K.-A. Lie, and J. R. Natvig. Solving the Euler Equations on Graphics Processing Units. In *Proceedings of the 6th International Conference on Computational Science*, volume 3994 of *Lecture Notes in Computer Science*, pages 220–227, 2006.

T.R. Hagen, M.O. Henriksen, J.M. Hjelmervik, and K.-A. Lie. How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a

High-Performance Computational Engine. In Geir Hasle, Knut-Andreas Lie, and Ewald Quak, editors, *Geometric Modelling, Numerical Simulation, and Optimization*, pages 211–264. Springer Berlin Heidelberg, 2007.