An MPI-CUDA implementation of an improved Roe method for two-layer shallow water systems

Marc de la Asunción[a], José M. Mantas[a], Manuel J. Castro[b], E. D. Fernández-Nieto[c]

[a] *Dpto. Lenguajes y Sistemas Informáticos, Universidad de Granada*
[b] *Dpto. Análisis Matemático, Universidad de Málaga*
[c] *Dpto. Matemática Aplicada I, Universidad de Sevilla*

Abstract

The numerical solution of two-layer shallow water systems is required to simulate accurately stratified fluids, which are ubiquitous in nature: they appear in atmospheric flows, ocean currents, oil spills, etc. Moreover, the implementation of the numerical schemes to solve these models in realistic scenarios imposes huge demands of computing power. In this paper, we tackle the acceleration of these simulations in triangular meshes by exploiting the combined power of several CUDA-enabled GPUs in a GPU cluster. For that purpose, an improvement of a path conservative Roe-type finite volume scheme which is specially suitable for GPU implementation is presented, and a distributed implementation of this scheme which uses CUDA and MPI to exploit the potential of a GPU cluster is developed. This implementation overlaps MPI communication with CPU-GPU memory transfers and GPU computation to increase efficiency. Several numerical experiments, performed on a cluster of modern CUDA-enabled GPUs, show the efficiency of the distributed solver.

Keywords: Shallow water simulation, GPU cluster computing, finite

volume schemes, unstructured meshes, CUDA, MPI

1. Introduction

The two-layer shallow water system has been used as the numerical model to simulate several phenomena related to stratified geophysical flows such as ocean currents, oil spills or tsunamis generated by underwater landslides. The simulation of these phenomena requires long lasting simulations in big domains, and the implementation of efficient numerical schemes to solve these problems on parallel platforms seems to be a suitable way of achieving the required performance in realistic applications.

A cost effective way of improving substantially the performance in these applications is the use of Graphics Processor Units (GPUs). These platforms make it possible to achieve speedups of an order of magnitude over a standard CPU in many applications and are growing in popularity [1, 2]. Moreover, several programming toolkits such as CUDA [3] have been developed to facilitate the programming of GPUs for general purpose applications.

There are previous works to port finite volume one-layer shallow water solvers to a GPU by using a graphics-specific programming language [4, 5, 6], but currently most of the proposals to simulate shallow flows on a single GPU are based on the CUDA programming model. A CUDA solver for one-layer system based on the first order finite volume scheme presented in [7] is described in [8] to deal with structured regular meshes. The extension of this CUDA solver for two-layer shallow water system is presented in [9]. There also exist proposals to implement, using CUDA-enabled GPUs, high order schemes to simulate one-layer systems [10, 11, 12] and to implement

first-order schemes for one and two-layer systems on triangular meshes [13].

Although the use of single GPU systems makes it possible to satisfy the performance requirements of several applications, many applications require huge meshes, large numbers of time steps and even real time accurate predictions (for instance, to approximate the effect of an unexpected oil spill). These characteristics suggest to combine the power of multiple GPUs.

One approach to use several GPUs is based on programming shared memory multi-GPU desktop systems. These platforms have been used in fluid dynamic [14] and shallow water [15, 16] simulations by combining shared memory programming primitives to manage threads in CPU and CUDA to program the GPU. However, this cost-effective approach only offer a reduced number of GPUs (2-8 GPUs) and more flexible systems are desirable.

A more flexible approach involves the use of clusters of GPU-enhanced computers where each node is equipped with a single GPU or with a multi-GPU system. The computation on GPU clusters could make it possible to scale the runtime reduction according to the number of GPUs (which can be easily increased). Thus, this approach is more flexible than using a multi-GPU desktop system and the memory limitations of a GPU-enhanced node can be overcome by suitably distributing the data among the nodes, enabling us to simulate significantly larger realistic models and with greater precision. The use of GPU clusters to accelerate intensive computations is gaining in popularity [17, 18, 19, 20, 21, 22]. In [23], a one-layer shallow water solver is implemented on a GPU cluster for tsunami simulation. Most of the proposals to exploit GPU clusters use MPI [24] to implement interprocess communication and CUDA [3] to program each GPU. A common way to

reduce the remote communication overhead in these systems consists in using non-blocking communication MPI functions to overlap the data transfers between nodes with GPU computation and CPU-GPU data transfers.

In this work, an implementation of an improved finite volume scheme is developed for a GPU cluster by using MPI and CUDA. This implementation incorporates an efficient management of the distributed unstructured mesh and mechanisms to overlap computation with communication.

The outline of the article is as follows: the next section describes the underlying mathematical model and presents an improvement of a first order Roe type finite volume scheme, called IR-Roe scheme. Section 3 describes a data parallel version of the IR-Roe scheme. Several implementations of the IR-Roe scheme and the classical Roe scheme [7, 25] are compared in Section 4. In the two next sections we describe a single and a multi-GPU distributed implementation, respectively, of the method for triangular meshes. Section 7 analyses the experimental results obtained when the implementations are applied to solve an internal dam break problem on a cluster of 4 GPUs. Finally, Section 8 summarizes the conclusions and presents the future work.

2. Numerical model

2.1. The two-layer shallow water system

Let us consider the system of equations governing the 2d flow of two superposed immiscible layers of shallow fluids in a subdomain $\Omega \subset \mathbb{R}^2$:

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

where $W = \begin{pmatrix} h_1 & q_{1,x} & q_{1,y} & h_2 & q_{2,x} & q_{2,y} \end{pmatrix}^T$,

$$F_1(W) = \begin{pmatrix} q_{1,x} & \frac{q_{1,x}^2}{h_1} + \frac{1}{2}gh_1^2 & \frac{q_{1,x}q_{1,y}}{h_1} & q_{2,x} & \frac{q_{2,x}^2}{h_2} + \frac{1}{2}gh_2^2 & \frac{q_{2,x}q_{2,y}}{h_2} \end{pmatrix}^T$$

$$F_2(W) = \begin{pmatrix} q_{1,y} & \frac{q_{1,x}q_{1,y}}{h_1} & \frac{q_{1,y}^2}{h_1} + \frac{1}{2}g\ h_1^2 & q_{2,y} & \frac{q_{2,x}q_{2,y}}{h_2} & \frac{q_{2,y}^2}{h_2} + \frac{1}{2}gh_2^2 \end{pmatrix}^T,$$

$$S_k(W) = \begin{pmatrix} 0 & gh_1(2-k) & gh_1(k-1) & 0 & gh_2(2-k) & gh_2(k-1) \end{pmatrix}^T,\ k = 1, 2,$$

$$B_k(W) = \begin{pmatrix} \mathbf{0} & \mathcal{P}_{1,k}(W) \\ r\mathcal{P}_{2,k}(W) & \mathbf{0} \end{pmatrix}\ \mathcal{P}_{l,k}(W) = \begin{pmatrix} 0 & 0 & 0 \\ -gh_l(2-k) & 0 & 0 \\ -gh_l(k-1) & 0 & 0 \end{pmatrix}\ l = 1, 2.$$

Index 1 in the unknowns makes reference to the upper layer and index 2 to the lower one; g is the gravity and $H(\boldsymbol{x})$, the depth function measured from a fixed level of reference; $r = \rho_1/\rho_2$ 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 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: $\boldsymbol{q}_i(\boldsymbol{x}, t) = \boldsymbol{u}_i(\boldsymbol{x}, t)h_i(\boldsymbol{x}, t), \quad i = 1, 2.$

Let us define the matrices $A_k(W) = J_k(W) - B_k(W)$, $k = 1, 2$ where $J_k(W) = \frac{\partial F_k}{\partial W}(W)$ are the Jacobians of the fluxes F_k, and we assume that (1) is strictly hyperbolic. Let us also remark that the system (1) verifies the property of invariance by rotations. Effectively, let us define

$$T_\eta = \begin{pmatrix} R_\eta & \mathbf{0} \\ \mathbf{0} & R_\eta \end{pmatrix}, \quad R_\eta = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \eta_x & \eta_y \\ 0 & -\eta_y & \eta_x \end{pmatrix}, \quad \forall \eta = (\eta_x, \eta_y) \in I\!R^2,$$

and let us denote $F_\eta(W) = F_1(W)\eta_x + F_2(W)\eta_y$, $\boldsymbol{B}(W) = (B_1(W), B_2(W))$, and $\boldsymbol{S}(W) = (S_1(W), S_2(W))$. Then

$$F_\eta(W) = T_\eta^{-1} F_1(T_\eta W),\ T_\eta \boldsymbol{B}(W)\cdot\eta = B_1(T_\eta W),\ T_\eta \boldsymbol{S}(W)\cdot\eta = S_1(T_\eta W). \tag{2}$$

Moreover, it is easy to check that $T_\eta W$ verifies the system

$$\partial_t(T_\eta W) + \partial_\eta F_1(T_\eta W) = B_1(T_\eta W)\partial_\eta W + S_1(T_\eta W)\partial_\eta H + Q_{\eta^\perp}, \tag{3}$$

where $Q_{\eta^\perp} = T_\eta\Big(-\partial_{\eta^\perp} F_{\eta^\perp}(W) + \boldsymbol{B}(W)\cdot\eta^\perp\partial_{\eta^\perp} W + \boldsymbol{S}(W)\cdot\eta^\perp\partial_{\eta^\perp} H\Big)$, being $\eta^\perp = (-\eta_y, \eta_x)$.

2.2. The IR-Roe Numerical Scheme

To discretize (1), the domain Ω is decomposed into L cells or finite volumes: $V_i \subset \mathbb{R}^2$. Here, it is assumed that the cells are triangles. Given a cell 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; Γ_{ij} the common edge of two neighboring cells V_i and V_j, and $|\Gamma_{ij}|$ its length; $\eta_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ the normal unit vector at Γ_{ij} pointing towards V_j; and W_i^n the constant approximation to the average of the solution in V_i at time t^n provided by the numerical scheme.

In order to advance in time, at every edge of the finite volume mesh, a projected Riemann problem along the normal direction is considered which is discretized by means of a 1D scheme. Note that the property of invariance by rotations allows us to define a 1D scheme for the 1D system defined in (3) where Q_{η^T} has been neglected as it retains the tangential components of the system (see [26] for more details). Moreover, the resulting 1D system can be split in two parts, one corresponding to the usual 1D two-layer shallow-water system, that corresponds to first, second, fourth and fifth equations

in (3) and two 'transport' equations related to the tangential component of the velocity fields, corresponding to the third and sixth equations of (3). This special structure of the 1D system allows us to split the 1D numerical scheme into two parts: an usual Roe scheme for the 1D two-layer system and a numerical treatment inspired in the HLLC method for the shallow water with pollutant transport introduced in [27], for the two transport equations. The resulting numerical scheme reads as follows:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|} \sum_{j \in \mathcal{N}_i} |\Gamma_{ij}| T_{\eta_{ij}}^{-1} F_{ij}^- \tag{4}$$

where $F_{ij}^- = \left[\begin{array}{ccccccc} \Phi_{\eta_{ij}}^{[1]} & \Phi_{\eta_{ij}}^{[2]} & \Phi_{\eta_{ij}}^{[1]} u_{1,\eta_{ij}^\perp}^* & \Phi_{\eta_{ij}}^{[3]} & \Phi_{\eta_{ij}}^{[4]} & \Phi_{\eta_{ij}}^{[3]} u_{2,\eta_{ij}^\perp}^* \end{array} \right]^T$, being $\Phi_{\eta_{ij}}^{[l]}$ the l-th component of the vector $\Phi_{\eta_{ij}} \in I\!R^4$ which is defined as follows:

$$\Phi_{\eta_{ij}} = \frac{1}{2} \mathcal{K}_{ij} (I - \mathrm{sgn}(\mathcal{D}_{ij})) \mathcal{K}_{ij}^{-1} \left[\mathcal{F}_1(W_{\eta_{ij},j}) - \mathcal{F}_1(W_{\eta_{ij},i}) - \mathcal{B}_{ij}(W_{\eta_{ij},j} - W_{\eta_{ij},i}) \right.$$

$$\left. - \mathcal{S}_{ij}(H_j - H_i) \right] + \mathcal{F}_1(W_{\eta_{ij},i}).$$

Here, $\mathcal{F}_1(W_{\eta_{ij},k}) = F_1(T_{\eta_{ij}} W_k)^{[1,2,4,5]} \quad (k = i, j)$, $\mathcal{B}_{ij}(W_{\eta_{ij,j}} - W_{\eta_{ij,i}}) = \big(B_{1,ij}(T_{\eta_{ij}}(W_j - W_i))\big)^{[1,2,4,5]}$, and $\mathcal{S}_{ij}(H_j - H_i) = \big(S_{1,ij}(H_j - H_i)\big)^{[1,2,4,5]}$ where $W^{[i_1,\cdots,i_s]}$ is the vector defined from vector W, using its i_1-th, ..., i_s-th components. I is the identity matrix, $\mathcal{K}_{ij}$ is the matrix whose columns are the eigenvectors of the matrix $\mathcal{A}_{ij}$, and $\mathrm{sgn}(\mathcal{D}_{ij})$ is the diagonal matrix whose coefficients are the signs of the eigenvalues of $\mathcal{A}_{ij}$, being

$$\mathcal{A}_{ij} = \left[\begin{array}{cccc} 0 & 1 & 0 & 0 \\ g h_{1,ij} - u_{1,\eta_{ij}}^2 & 2u_{1,\eta_{ij}} & g h_{1,ij} & 0 \\ 0 & 0 & 0 & 1 \\ r g h_{2,ij} & 0 & g h_{2,ij} - u_{2,\eta_{ij}}^2 & 2u_{2,\eta_{ij}} \end{array} \right] \begin{array}{c} h_{k,ij} = \frac{h_{k,i} + h_{k,j}}{2}, \\ u_{k,\eta_{ij}} = \boldsymbol{u}_{k,ij} \cdot \eta_{ij}, \\ k = 1, 2. \end{array}$$

$u^*_{k,\eta_{ij}^\perp}(k=1,2)$ are defined as follows: $u^*_{k,\eta_{ij}^\perp} = \begin{cases} \frac{q_{k,i}\cdot\eta_{ij}^\perp}{h_{k,i}} & \text{if } (\Phi_{\eta_{ij}})^{[2k-1]} > 0 \\ \frac{q_{k,j}\cdot\eta_{ij}^\perp}{h_{k,j}} & \text{otherwise} \end{cases}$

$\Phi_{\eta_{ij}}$ is the 1D numerical Roe flux associated to the 1D system defined by the 1-st, 2-nd, 4-th and 5-th equations of system (3) and $(\Phi^{[1]}_{\eta_{ij}} u^*_{1,\eta_{ij}^\perp}, \Phi^{[3]}_{\eta_{ij}} u^*_{2,\eta_{ij}^\perp})^T$ is the numerical flux associated to the 3-rd and 6-th equations of system (3) where, in both cases, the term $Q_{\eta_{ij}^\perp}$ has been neglected.

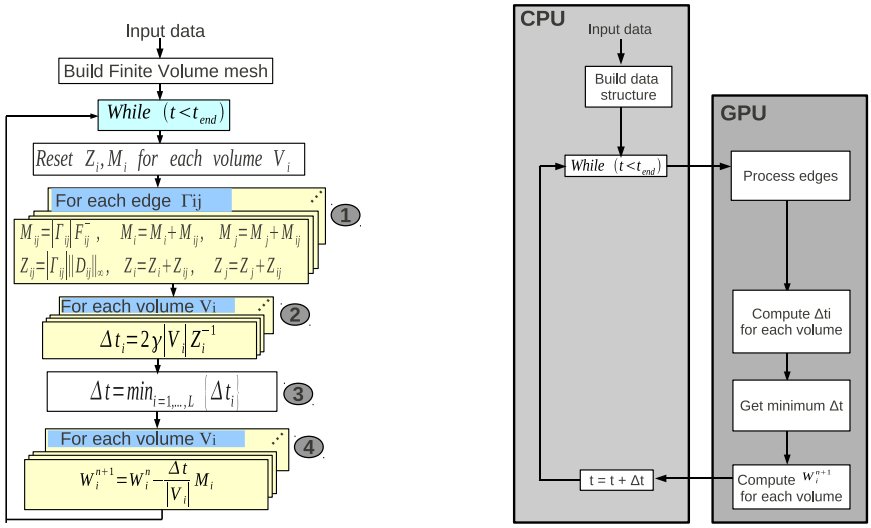
A CFL condition must be imposed to ensure stability of both schemes:

$$\frac{1}{2}\frac{\Delta t}{V_i}\sum_{j\in\mathcal{N}_i} |\Gamma_{ij}\|\mathcal{D}_{ij}\|_\infty \leq \gamma, \quad 1 \leq i \leq L, \text{ with } 0 < \gamma \leq 1. \tag{5}$$

As in the case of systems of conservation laws, when sonic rarefaction waves appear it is necessary to modify the numerical scheme to get entropy-satisfying solutions. For instance, the Harten-Hyman entropy fix technique [28] can be easily adapted here. Let us also remark that the scheme is path-conservative in the sense introduced by Pares in [29] and [25]. It is well-balanced for stationary solutions corresponding to water at rest. More general results concerning the consistency and well-balanced properties of Roe schemes have been studied in [25] and [30].

3. Parallelization of the scheme

Figure 1a shows a graphical description of the parallel algorithm, obtained from the description of the IR-Roe numerical scheme given in Section 2. The main calculation phases are identified with circled numbers, and the main sources of data parallelism are represented with overlapping rectangles.



(a) Parallelism sources (b) CUDA implementation

Figure 1: Main calculation phases

Initially, the finite volume mesh is built. Then the time stepping process is repeated until the final simulation time is reached. At the $(n+1)$-th time step, Equation (4) must be evaluated to update the state of each cell.

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

1. Edge-based calculations: This is the most costly phase of the algorithm. It involves two calculations for each edge Γ_{ij} communicating two cells V_i and V_j $(i, j \in \{1, \ldots, L\})$: **a)** The computation of the contribution $M_{ij} =\mid \Gamma_{ij} \mid F_{ij}{}^{-}$ (a 6×1 vector) to the sums M_i and M_j associated to V_i and V_j (see Eq. (4)), and **b)** the computation of the contribution $Z_{ij} =\mid \Gamma_{ij} \mid\mid\mid \mathcal{D}_{ij}^n \mid\mid_\infty$ to the sums Z_i and Z_j associated to V_i and V_j (see Eq. (5)).

2. Computation of the local Δt for each volume: For each volume

V_i, the local Δt_i is obtained as follows (see Eq. (5)): $\Delta t_i = 2\gamma\,|V_i|\,Z_i^{-1}$.

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.

Since the numerical scheme exhibits a high degree of potential data parallelism, it is good candidate to be implemented on CUDA architectures.

4. Roe Schemes Comparison

In this section we will compare the efficiency of several implementations of the IR-Roe method and the classical Roe scheme introduced in [7].

We consider an internal circular dambreak problem in the $[-5,5]\times[-5,5]$ rectangular domain. The depth function is $H(x,y) = 5$, and the initial condition is: $W_i^0(x,y) = (h_1(x,y),\ 0,\ 0,\ h_2(x,y),\ 0,\ 0)^{\mathrm{T}}$, where:

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

The numerical scheme is run for several triangular meshes (see Table 1). Simulation time interval is $[0, 0.1]$, CFL parameter is $\gamma = 0.9$, $r = 0.998$ and wall boundary conditions ($q_1 \cdot \boldsymbol{\eta} = 0,\ q_2 \cdot \boldsymbol{\eta} = 0$) are considered.

We have implemented two programs for each Roe method: a serial and a quadcore CPU version. The latter is a parallelization of the serial CPU version using OpenMP [31]. Both programs have been implemented in C++ using double precision and the Eigen library [32] for operating with matrices.

The IR-Roe method requires to find the eigenvectors of 4×4 matrices, but these can be calculated directly by evaluating formulas and single precision

Number of cells	Classical Roe		IR-Roe			
	1 core	4 cores	1 core		4 cores	
4016	0.61	0.16	0.11	(5.5)	0.031	(5.2)
16040	4.84	1.27	0.88	(5.5)	0.27	(4.7)
64052	41.38	11.18	7.76	(5.3)	2.50	(4.5)
256576	382.5	103.6	63.87	(6.0)	20.30	(5.1)
1001898	4402.0	1282.8	494.7	(8.9)	154.9	(8.3)
2000608	14207.9	4878.3	1469.4	(9.7)	452.9	(10.8)
3000948	25716.9	8782.7	2697.8	(9.5)	830.5	(10.6)

Table 1: CPU execution times in seconds for both Roe methods.

arithmetic is sufficient. On the other hand, the classical Roe scheme involves to find the eigenvectors of 6×6 matrices by using an iterative method which requires double precision arithmetic to ensure numerical stability.

All the programs were executed on a Core i7 920 with 4 GB RAM and the GNU C++ compiler was used with `-O3 -DNDEBUG` switches. Table 1 shows the execution times in seconds and the speedup obtained with the IR-Roe method with respect to the classical Roe implementation (in parenthesis).

As it can be seen, the IR-Roe method clearly outperforms the classical Roe method in all cases, obtaining a speedup of 10 for meshes having more than $2 \cdot 10^6$ volumes. Moreover, this scheme is more suitable to be ported to CUDA-enabled GPUs because it does not need to use double precision floating point arithmetic and the computational demand of each GPU thread is lower. Since the matrices and vectors which are used are smaller than in the classical Roe scheme, the register usage is also smaller.

5. CUDA Implementation of the IR-Roe method

The CUDA implementation of the algorithm exposed in Section 3 is a variant of the implementation described in [13], Section 7.3. The general steps of the implementation are depicted in Figure 1b. Each step executed on the GPU is assigned to a CUDA kernel and corresponds to a calculation phase described in Section 3. Next, we briefly describe each step:

- Build data structure: Volume data is stored in two arrays of L `float4` elements as 1D textures, where each element contains the data (state, depth and area) of a cell. We have used textures because each edge (thread) only needs the data of adjacent cell and texture memory is especially suited for each thread to access its closer environment in texture memory. We derived CUDA implementations of the classical Roe scheme using textures and shared memory in structured meshes [9] and better performance was obtained by exploiting the texture cache. The ordering of the cells is given by the output of the `Triangle` program [33], used to generate the triangle mesh, which numbers the triangles as they are generated by the triangulation algorithm. Edge data is stored in two arrays in global memory with a size equal to the number of edges: an array of `float2` elements for storing the normals, and another array of `int4` elements for storing, for each edge, the positions of the neighboring volumes in the volume textures and the two accumulators where the edge must write its contributions to the neighboring volumes.

- Process edges: In this step each thread represents an edge, and computes the contribution of the edge to their adjacent volumes.

The threads contribute to a particular cell by means of six accumulators, each one being an array of L `float4` elements. Let us call the accumulators

1-1, 1-2, 2-1, 2-2, 3-1 and 3-2. Each element of 1-1, 2-1 and 3-1 stores the contributions of the edges to the layer 1 and to the local Δt of W_i, while each element of 1-2, 2-2 and 3-2 stores the contributions of the edges to the layer 2 of W_i. This kernel is computationally more efficient than that proposed in [13] (which implements the edge processing for the classical Roe scheme) and results more suitable for GPU implementation because it does not need to perform any double precision calculation and its matrix and vector operations mainly use arguments with dimensions 4×4 and 4×1.

- 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

- Get minimum Δt: This step finds the minimum of the local Δt_i of the volumes by applying the most optimized kernel of the reduction sample included in the CUDA Software Development Kit [34].

- 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. The first 3 elements of M_i are obtained by summing 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 are obtained by summing the three 3×1 vectors stored in the equivalent positions 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.

6. Implementation on a GPU cluster

In this section a multi-GPU extension of the CUDA implementation detailed in Section 5 is proposed. Basically, the triangular mesh is divided into

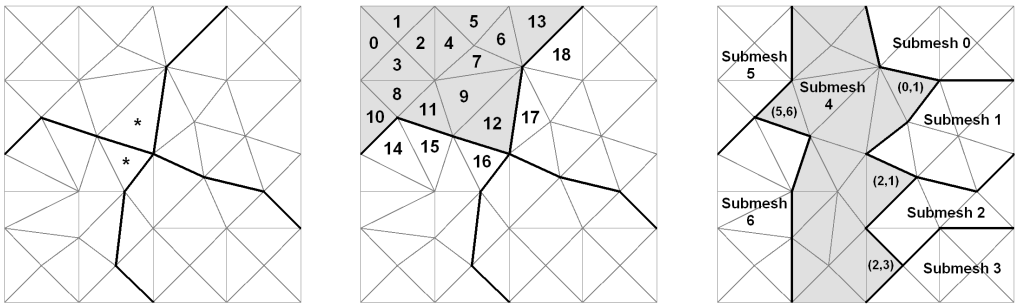
several submeshes and each submesh is assigned to a CPU process, which, in turn, uses a GPU to perform the computations related to its submesh. We use MPI [24] for the communication between processes. Next we describe how the data of a particular submesh is created and stored in GPU memory.

6.1. Creation of the Submesh Data

We consider two types of submeshes: those that have cells that must be sent to two different MPI processes (i.e. submeshes) in each iteration (**type 1**), and those that do not have cells that fulfill the former condition (**type 2**). For example, in Figure 2a the two left submeshes are of type 1, and the two right submeshes belong to type 2. The cells that must be sent to two MPI processes are noted with an asterisk. Both types of submesh use the same data structure for its edges and cells, explained in Section 5, but now the arrays of cells are divided into three blocks:

1. Firstly, the non-communication cells of the submesh are stored (a communication cell is a cell that has an adjacent edge to another submesh).
2. Secondly, the communication cells of the submesh are stored in the array. In turn, these cells are divided into groups of cells that are adjacent to a particular submesh.
3. Finally, the communication cells of other submeshes that are adjacent to our submesh are stored. In turn, these cells are divided into groups of cells that belong to a particular submesh.

Figure 2b shows a possible cell indexation of the cell data used by the gray submesh. Block 1 is formed by cells 0-9, block 2 consists of cells 10-13, and cells 14-18 belong to block 3.



(a) Types of submeshes (b) Volume indexation (c) Sequence of pairs

Figure 2: Submeshes.

6.2. Creation of Data in Submeshes of Type 1

The creation of data in submeshes of type 1 is more complicated because we must arrange the communication cells of the submesh so that all the cells that are adjacent to a particular submesh appear consecutively in the array. For example, in Figure 2b, cells 10-12 are sent to the lower submesh, while cells 12 and 13 are sent to the right submesh, thus overlapping the sendings.

In order to perform this arrangement, firstly we build a list of communication cells for each adjacent submesh. For example, in Figure 2b we would have two lists: [10, 11, 12] and [12, 13].

Now, for each communication cell of the submesh that must be sent to two MPI processes, we build a pair (p_1, p_2), meaning that the cell must be sent to processes p_1 and p_2. Figure 2c shows an example centered on submesh 4, where all the pairs are specified. Once all the pairs have been built, we perform a reordering of them (and their elements if necessary) so that we get a list of consecutive processes. In Figure 2c, the pairs are reordered obtaining: $(0, 1)$, $(1, 2)$, $(2, 3)$ and $(5, 6)$. This gives the consecutive list of

processes 0, 1, 2, 3, 5 and 6. Now we carry out the adequate swaps: **1)** In the list storing the cells that are adjacent to submesh 0, we put the cell shared with submesh 1 at the end. **2)** In the list storing the cells that are adjacent to submesh 1, we put the cell shared with submesh 0 at the start, and the cell shared with submesh 2 at the end. **3)** We continue processing the list in the same way until it finishes.

Finally we join all the lists of communication cells adequately to get the definitive block of communication cells.

Since a submesh must know the ordering of the communication cells that receives from another submesh, at this point each submesh send to their adjacent submeshes all the position swappings of the lists that it has performed during the creation of its data. Note also that this algorithm does not work when a submesh has two cells that must be sent to the same submeshes, but we can always perform a mesh partitioning where this does not occur.

6.3. Multi-GPU Code

We have implemented two versions of the multi-GPU algorithm: one with blocking MPI sends and receives, and another one which overlaps MPI communication with CPU-GPU memory transfers and kernel computation.

Algorithm 1 shows the general steps of the non-overlapping implementation. In lines 3-6 we send to each adjacent submesh the communication cells that are adjacent to it. Then, in lines 7-10, we receive from the same submeshes their communication cells that are adjacent to our submesh. We have used bufferized MPI send operations with a given buffer to avoid deadlocks with big meshes. Lines 11-12 copy the received communication cells to

Algorithm 1 Non-overlapping multi-GPU algorithm

```
1: n ← number of adjacent submeshes
2: while (t < t_end) do
3:    for i = 1 to n do
4:       MPI_Bsend(Layer 1 of comm. cells to adjacent submesh i)
5:       MPI_Bsend(Layer 2 of comm. cells to adjacent submesh i)
6:    end for
7:    for i = 1 to n do
8:       MPI_Recv(Layer 1 of comm. cells from adjacent submesh i)
9:       MPI_Recv(Layer 2 of comm. cells from adjacent submesh i)
10:   end for
11:   CudaMemcpy(Layer 1 of comm. cells from host to device)
12:   CudaMemcpy(Layer 2 of comm. cells from host to device)
13:   processEdges<<<grid, block>>>(...)
14:   computeDeltaTVolumes<<<grid, block>>>(...)
15:   Δt ← getMinimumDeltaT(...)
16:   MPI_Allreduce(Δt, min Δt, ...)
17:   computeVolumeStates<<<grid, block>>>(...)
18:   CudaMemcpy(Layer 1 of comm. cells from device to host)
19:   CudaMemcpy(Layer 2 of comm. cells from device to host)
20:   t ← t + min Δt
21: end while
```

GPU memory. In line 16 a MPI reduction is performed to obtain the global minimum Δt in all the MPI processes. Lines 18-19 copy the new states of the communication cells of our submesh from device to host.

Algorithm 2 shows the general steps of the overlapping implementation. Lines 3-6 (the reception of the communication cells from the adjacent submeshes) overlap with lines 7-8 (the copy of the new states of our communication cells from device to host) and with line 13 (the processing of the non-communication edges, since these edges do not need external data to be processed). Lines 9-12 (the sending to each adjacent submesh of the cells adjacent to it) also overlap with line 13. In lines 14-15, we wait for the

Algorithm 2 Overlapping multi-GPU algorithm

```
1: n ← number of adjacent submeshes
2: while (t < t_end) do
3:    for i = 1 to n do
4:       MPI_Irecv(Layer 1 of comm. cells from adjacent submesh i)
5:       MPI_Irecv(Layer 2 of comm. cells from adjacent submesh i)
6:    end for
7:    CudaMemcpy(Layer 1 of comm. cells from device to host)
8:    CudaMemcpy(Layer 2 of comm. cells from device to host)
9:    for i = 1 to n do
10:      MPI_Isend(Layer 1 of comm. cells to adjacent submesh i)
11:      MPI_Isend(Layer 2 of comm. cells to adjacent submesh i)
12:   end for
13:   processEdges<<<grid, block>>>(Non-communication edges)
14:   MPI_Waitall (Layer 1 of comm. cells from adjacent submeshes)
15:   MPI_Waitall (Layer 2 of comm. cells from adjacent submeshes)
16:   CudaMemcpy(Layer 1 of comm. cells from host to device)
17:   CudaMemcpy(Layer 2 of comm. cells from host to device)
18:   processEdges<<<grid, block>>>(Communication edges)
19:   computeDeltaTVolumes<<<grid, block>>>(...)
20:   Δt ← getMinimumDeltaT(...)
21:   MPI_Allreduce(Δt, min Δt, ...)
22:   computeVolumeStates<<<grid, block>>>(...)
23:   t ← t + min Δt
24: end while
```

communication cells of the adjacent submeshes to arrive, and in lines 16-17 we copy them to GPU. In line 18 only the communication edges are processed.

7. Experimental Results

In this section we will test the single and multi-GPU implementations described in Sections 5 and 6, respectively. The test problem and the parameters are the same that were used in Section 4.

We have used the Chaco software [35] to divide a mesh into equally sized

Number of cells	GTX 480	2 GTX 480		4 GTX 480	
		Non-Overlap	Overlap	Non-Overlap	Overlap
4016	0.0068	0.0085	0.010	0.010	0.013
16040	0.032	0.027	0.029	0.023	0.025
64052	0.21	0.13	0.13	0.10	0.093
256576	1.49	0.85	0.81	0.54	0.46
1001898	11.05	6.12	5.74	3.92	3.15
2000608	32.08	16.84	16.32	9.10	8.37
3000948	58.53	31.92	29.66	20.72	15.14

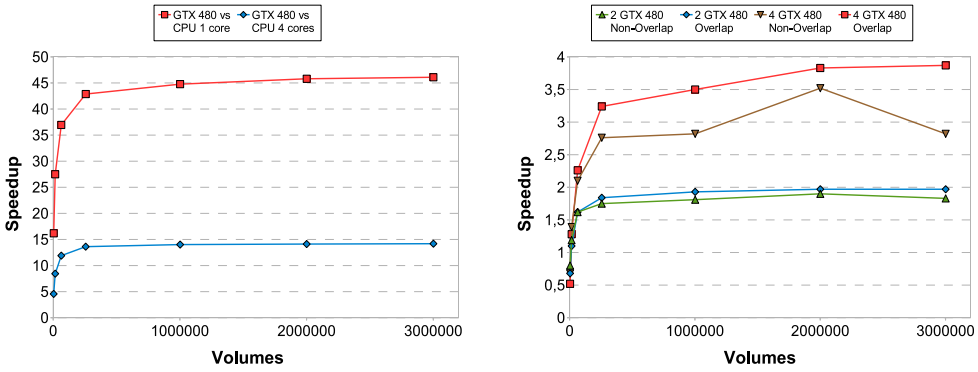
Table 2: GPU execution times in seconds for the IR-Roe method.

submeshes, the OpenMPI implementation [36] and the GNU compiler. All the programs were executed in a cluster formed by four Intel Xeon servers with 8 GB RAM each one, connected with a Gigabit Ethernet switch. Graphics cards used were four GeForce GTX 480. Table 2 shows the execution times in seconds for all the meshes and number of GPUs. Figure 3a shows graphically the speedups obtained with the single GPU program executed on a GTX 480 with respect to the CPU versions of the IR-Roe method used in Section 4 (using the same executables and CPU employed in Section 4).

Number of cells	$T_{send\&receive}$		$T_{EdgeProcessing}$		$T_{Alg1} - T_{Alg2}$
	Alg1	Alg2	Alg1	Alg2	
1001898	0.95	0.05	2.78	2.9	0.77
2000608	0.95	0.04	7.72	7.81	0.73
3000948	5.59	0.01	14.16	14.35	5.58

Table 3: Times for several phases on 4 GPUs and big meshes.

Figure 3b shows the speedups obtained with the multi-GPU implementations with respect to one GTX 480. As it can be seen, using a GTX 480, for



(a) GTX 480 speedup with respect to serial and quadcore CPU versions

(b) Multi-GPU speedup with respect to one GTX 480

Figure 3: Speedups obtained for one and several GPUs.

meshes with more than one million cells, we have reached a speedup of 45 and 14 with respect to monocore and quadcore CPU versions, respectively. As expected, the overlapping multi-GPU implementation outperforms the non-overlapping version despite having one additional kernel launch. In order to make the benefits of the overlapping clearer, Table 3 shows the times spent in executing MPI send and receive operations ($T_{send\&receive}$), edge processing ($T_{EdgeProcessing}$) and the runtime difference for both implementations on 4 GPUs using meshes with more than 10^6 cells. As can be seen, the runtime difference among both versions is very close to $T_{send\&receive}$ in Algorithm 1 (lines 3 to 10). Therefore, the cost associated to exchange of communication cell data in Algorithm 2 is masked in a great deal with the processing of non-communication edges.

Figure 4 shows the way in which the speedup (with respect to the single GPU version) of the overlapping version changes with the number of GPUs for all the mesh sizes. As can be seen, the bigger the mesh size is, the closer

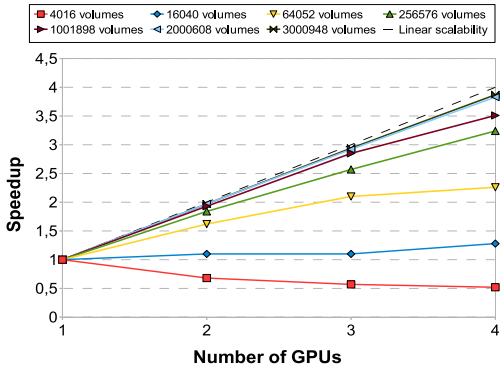


Figure 4: Multi-GPU speedup vs. one GPU for the overlapping version.

the scaling is to the linear one. The scaling is not good for small meshes because the small size of the submehes prevents each GPU from obtainining the best performance and the overhead due to the remote communication and GPU-CPU data transfer is proportionally much higher than with big meshes. However, for meshes with more than 10^6 cells, the weak and strong scaling reached are close to the linear scaling for up to four GPUs.

8. Conclusions and future work

In this paper we have presented an improvement of a first order well-balanced Roe-type finite cell solver for two-layer shallow water system. This numerical scheme has proved to be computationally more efficient than the classical Roe scheme and is more suitable to be implemented in modern CUDA-enabled GPUs than the classical Roe scheme. A multi-GPU distributed implementation of this scheme that works on triangular meshes and overlap remote communications with GPU computation has been implemented using MPI and CUDA. Numerical experiments carried out on a GPU cluster have shown the efficiency of this solver, obtaining close to linear

weak and strong scaling for up to four GPUs on unstructured meshes with more than 10^6 cells. The effects of the overlapping to reduce the remote communication overhead in this application have shown to be very relevant.

As further work, we propose to extend the method to enable high order numerical schemes and to integrate a dynamic load balancing strategy (which is necessary in problems where the computational load for each spatial subdomain could vary dramatically).

Acknowledgements

The authors acknowledge partial support from the DGI-MEC projects MTM2008-06349-C03-03, MTM2009-11923 and MTM2009-07719.

References

[1] M. Rumpf, R. Strzodka, Graphics Processor Units: New Prospects for Parallel Computing, in: Numerical Solution of Partial Differential Equations on Parallel Computers, Vol. 51 of Lecture Notes in Computational Science and Engineering, Springer, 2005, pp. 89–134.

[2] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.

[3] NVIDIA Corporation, NVIDIA CUDA C Programming Guide 3.2, 2010.

[4] T. Hagen, J. Hjelmervik, K.-A. Lie, J. Natvig, M. O. Henriksen, Visual simulation of shallow-water waves, Simulation Modelling Practice and Theory 13 (8) (2005) 716–726, Programmable Graphics Hardware.

[5] M. Lastra, J. M. Mantas, C. Ureña, M. J. Castro, J. A. García-Rodríguez, Simulation of shallow-water systems using graphics processing units, Mathematics and Computers in Simulation 80 (3) (2009) 598–618.

[6] W.-Y. Liang, T.-J. Hsieh, M. T. Satria, Y.-L. Chang, J.-P. Fang, C.-C. Chen, C.-C. Han, A GPU-Based Simulation of Tsunami Propagation and Inundation, in: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 593–603.

[7] M. Castro, J. García-Rodríguez, J. González-Vida, C. Parés, A parallel 2d finite volume scheme for solving systems of balance laws with non-conservative products: Application to shallow flows, Computer Methods in Applied Mechanics and Engineering 195 (19-22) (2006) 2788–2815.

[8] M. de la Asunción, J. M. Mantas, M. J. Castro, Simulation of one-layer shallow water systems on multicore and CUDA architectures, The Journal of Supercomputing (2010) 1–9.

[9] M. de la Asunción, J. M. Mantas, M. J. Castro, Programming CUDA-based GPUs to simulate two-layer shallow water flows, in: Euro-Par 2010 - Parallel Processing, Vol. 6272 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2010, pp. 353–364.

[10] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, GPU computing for shallow water flow simulation based on finite volume schemes, Bol. Soc. Esp. Mat. Apl. 50 (2010) 27–45.

[11] A. Brodtkorb, T. Hagen, K.-A. Lie, J. Natvig, Simulation and visualization of the saint-venant system using GPUs, Computing and Visualization in Science (2011) 1–13.

[12] J. Gallardo, S. Ortega, M. de la Asunción, J. M. Mantas, Two-dimensional compact third-order polynomial reconstructions. Solving nonconservative hyperbolic systems using GPUs, Journal of Scientific Computing (2011) 1–23.

[13] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, J. M. Gallardo, GPU computing for shallow water flow simulation based on finite volume schemes, Comptes Rendus Mécanique 339 (2-3) (2011) 165–184, High Performance Computing.

[14] J. Thibault, I. Senocak, Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms, The Journal of Supercomputing (2010) 1–27.

[15] M. Geveler, D. Ribbrock, S. Mallach, D. G A Simulation Suite for Lattice-Boltzmann based Real-Time CFD Applications Exploiting Multi-Level Parallelism on modern Multi- and Many-Core Architectures, Journal of Computational Science In Press, Accepted Manuscript.

[16] M. L. Saetra, A. R. Brodtkorb, Shallow water simulations on multiple GPUs, Proceedings of the Para 2010 Conference, Lecture Notes in Computer Science (2010) Accepted for publication.

[17] D. Komatitsch, G. Erlebacher, D. Göddeke, D. Michéa, High-order

finite-element seismic wave propagation modeling with MPI on a large GPU cluster, J. Comput. Phys. 229 (2010) 7692–7714.

[18] D. Komatitsch, Fluid-solid coupling on a cluster of GPU graphics cards for seismic wave propagation, Comptes Rendus Mécanique 339 (2-3) (2011) 125–135, High Performance Computing.

[19] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, GPU Cluster for High Performance Computing, in: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04, IEEE Computer Society, Washington, DC, USA, 2004.

[20] Y. Zhang, F. Mueller, X. Cui, T. Potok, Data-intensive document clustering on graphics processing unit (GPU) clusters, J. Parallel Distrib. Comput. 71 (2011) 211–224.

[21] R. Abdelkhalek, H. Calendra, O. Coulaud, J. Roman, G. Latu, Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster, in: The 2009 High Performance Computing & Simulation - HPCS'09, Leipzig Allemagne, 2009, Best Paper Award at HPCS'09 Total.

[22] D. A. Jacobsen, J. C. Thibault, I. Senocak, An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters, in: The 2009 High Performance Computing & Simulation - HPCS'09, Orlando, FL., 2010.

[23] M. Acuña, T. Aoki, Real-time tsunami simulation on multi-node GPU cluster, Supercomputing (2009) [Poster].

[24] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, Univ. of Tennessee, Knoxville, Tennessee.

[25] M. Castro, E. Fernández, A. Ferreiro, A. García, C. Parés, High order extension of Roe schemes for two dimensional nonconservative hyperbolic systems, J. Sci. Comput. 39 (2009) 67–114.

[26] E. D. Fernández-Nieto, Modelling an numerical simulation of submarine sediment shallow flows: transport and avalanches, Bol. Soc. Esp. Mat. Apl. 49.

[27] E. D. Fernández-Nieto, D. Bresch, J. Monnier, A consistent intermediate wave speed for a well-balanced HLLC solver, Comptes Rendus Mathematique 346 (13-14) (2008) 795 – 800.

[28] J. H. A. Harten, Self-adjusting grid methods for one-dimensional hyperbolic conservation laws, J. Comp. Phys. 50 (1983) 235–269.

[29] C. Parés, High order extension of Roe schemes for two dimensional nonconservative hyperbolic systems, SIAM J. Num. Anal. 44 (2006) 300–321.

[30] C. Parés, M. Castro, On the well-balance property of Roe's method for nonconservative hyperbolic systems. applications to shallow-water systems, M2AN 38 (5) (2004) 821–852.

[31] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, 2007.

[32] G. Guennebaud, B. Jacob, et al., Eigen v2.0.15, http://eigen.tuxfamily.org (2010).

[33] J. R. Shewchuk, Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, Lecture Notes in Computer Science 1148.

[34] NVIDIA Corporation, CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html.

[35] B. Hendrickson, R. Leland, The Chaco User's Guide: Version 2.0, Sandia Tech Report SAND94–2692. Sandia National Laboratories.

[36] E. G. et. al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004, pp. 97–104.