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, . . . 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 atmospheric flows, ocean currents, oil spills or tsunamis generated by underwater landslides. The simulation of these phenomena gives to place to very long lasting simulations in big computational domains. Moreover, some of these phenomena (tsunami propagation or oil spills) could require real time calculation. Therefore, extremely efficient numerical schemes and implementations are needed to be able to analyze those problems in practical execution times. Since the numerical schemes to solve shallow water systems usually exhibit a high degree of potential parallelism, the development of parallel versions of these schemes for high performance platforms seems to be a suitable way of achieving the required performance in realistic applications.

A cost effective way of obtaining a substantially higher performance in these applications consists in using Graphics Processor Units (GPUs). These architectures make it possible to obtain performances that are orders of magnitude faster than a standard CPU and are growing in popularity among the scientific and engineering community [1, 2]. Moreover, several GPU programming toolkits such as CUDA [3] have been developed to facilitate the programming of GPUs for general purpose applications.

There are previous proposals 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 which are not computationally expensive, this situation is not frequent in most realistic applications. Many applications require to handle huge meshes and large number of time steps. Moreover, in some applications real time accurate predictions (for instance, to approximate the effect of an unexpected oil spill) could be required. The characteristics of these applications suggest to combine the power of multiple GPUs to satisfy the performance requirements.

One approach to use several GPUs in these problems is based on programming shared memory multi-GPU desktop systems. These platforms have been used to accelerate considerably fluid dynamic [14] and shallow water [15] simulations by combining shared memory programming primitives to manage threads in CPU and CUDA to program the GPU. Although this is a cost-effective approach, these platforms only offer a reduced number of GPUs and more flexible systems are desirable to answer to the growing performance requirements of many realistic applications.

A more flexible approach to obtain the required performance involves to use 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 reduction in execution time according to the number of GPUs (which can be easily increased). Therefore, 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 data intensive computations is gaining in popularity [16, 17, 18, 19]. In [20], a scheme to solve one-layer shallow water systems is implemented on a GPU cluster for real-time tsunami simulation. Most of the proposals to exploit GPU clusters in scientific computing use MPI [21] to implement the communication among the processes of the distributed system and CUDA [3] to program the GPU (or GPUs) associated to each node. A common way to reduce the remote communication overhead in these distributed implementations consists in using non-blocking communication MPI functions to overlap the data transfers between nodes of the cluster with the GPU computation and the CPU-GPU data transfers.

This work deals with the acceleration of the numerical solution of two-layer shallow water systems by exploiting the parallelization of an improved finite volume scheme for unstructured meshes on GPU clusters. For that purpose, a distributed implementation of this scheme for a GPU cluster is developed 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. The efficiency of several implementations of the IR-Roe scheme and the classical Roe scheme [7, 22] is 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 using the CUDA framework. Section 7 shows the experimental results obtained when the implementations are applied to solve an internal dam break problem on a cluster of 4 NVIDIA Fermi GPUs. Finally, Section 8 summarizes the main conclusions and presents the future work.

2. An efficient numerical scheme

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

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

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

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

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

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

$$T_\eta = \begin{pmatrix} R_\eta & \mathbf{0} \\ \mathbf{0} & R_\eta \end{pmatrix}, \quad R_\eta = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \eta_x & \eta_y \\ 0 & -\eta_y & \eta_x \end{pmatrix},$$

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

2.2. The IR-Roe Numerical Scheme

To discretize (1) the computational domain Ω is decomposed into cells or finite volumes: $V_i \subset \mathbb{R}^2$. Here, it is assumed that the cells are triangles. Let us denote by L the number of triangles of the mesh.

Given a finite volume V_i, $|V_i|$ will represent its area; $N_i \in \mathbb{R}^2$ its center; $\mathcal{N}_i$ the set of indexes j such that V_j is a neighbor of V_i; Γ_{ij} the common edge of two neighboring triangles V_i and V_j, and $|\Gamma_{ij}|$ its length; $\eta_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ the normal unit vector at the edge Γ_{ij} pointing towards the triangle V_j; and W_i^n the constant approximation to the average of the solution in the triangle V_i at time t^n provided by the numerical scheme.

Let us now briefly describe the Roe type scheme for system (1) that can be defined taking into account the property of invariance by rotations [23]:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|} \sum_{j \in \mathcal{N}_i} |\Gamma_{ij}| \mathcal{F}_{ij}^{IR-ROE-} \tag{4}$$

where $\mathcal{F}_{ij}^{IR-ROE-}$ is defined as follows:

1. Let us define $W_\eta = [h_1 \ q_{1,\eta} \ h_2 \ q_{2,\eta}]^T = T_\eta(W)_{[1,2,4,5]}$, and $W_{\eta^\perp} = [q_{1,\eta^\perp} \ q_{2,\eta^\perp}]^T = T_\eta(W)_{[3,6]}$, where $W_{[i_1,\cdots,i_s]}$ is the vector defined from vector W, using its i_1-th, ..., i_s-th components.
2. Let $\Phi_{\eta_{ij}}^-$ be the 1D numerical Roe flux associated to the 1D two-layer shallow-water system defined by the 1-st, 2-nd, 4-th and 5-th equations of system (3) where the term $Q_{\eta_{ij}^\perp}$ has been neglected:

$$\Phi_{\eta_{ij}}^- = \mathcal{P}_{ij}^-(\mathcal{F}_1(W_{\eta_{ij},j}) - \mathcal{F}_1(W_{\eta_{ij},i}) - \mathcal{B}_{ij}(W_{\eta_{ij},j} - W_{\eta_{ij},i}) - \mathcal{S}_{ij}(H_j - H_i)) \\ + \mathcal{F}_1(W_{\eta_{ij},i}).$$

where $\mathcal{F}_1(W_{\eta_{ij}}) = F_1(T_{\eta_{ij}}W)_{[1,2,4,5]}$, $\mathcal{B}_{ij}(W_{\eta_{ij,j}} - W_{\eta_{ij,i}}) = \big(B_{1,ij}(T_{\eta_{ij}}(W_j - W_i))\big)_{[1,2,4,5]}$, $\mathcal{S}_{ij}(H_j - H_i) = \big(S_{1,ij}(H_j - H_i)\big)_{[1,2,4,5]}$.

Finally, $\mathcal{P}^-_{ij} = \frac{1}{2}\mathcal{K}_{ij}(I - \mathrm{sgn}(\mathcal{D}_{ij}))\mathcal{K}^{-1}_{ij}$, where 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

$$\mathcal{A}_{ij} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ gh_{1,ij} - u^2_{1,\eta_{ij}} & 2u_{1,\eta_{ij}} & gh_{1,ij} & 0 \\ 0 & 0 & 0 & 1 \\ rgh_{2,ij} & 0 & gh_{2,ij} - u^2_{2,\eta_{ij}} & 2u_{2,\eta_{ij}} \end{bmatrix}$$

with $u_{k,\eta_{ij}} = \boldsymbol{u}_{k,ij} \cdot \eta_{ij}$, $k = 1,2$ and $h_{k,ij} = \frac{h_{k,i}+h_{k,j}}{2}$, where $u_{k,l,ij} = \frac{\sqrt{h_{k,i}}u_{k,l,i}+\sqrt{h_{k,j}}u_{k,l,j}}{\sqrt{h_{k,i}}+\sqrt{h_{k,j}}}$, $k = 1,2$, $l = x, y$.

3. Let us define $\Phi^-_{\eta^\perp_{ij}} = \begin{bmatrix} (\Phi^-_{\eta_{ij}})_{[1]}u^*_{1,\eta^\perp_{ij}} & (\Phi^-_{\eta_{ij}})_{[3]}u^*_{2,\eta^\perp_{ij}} \end{bmatrix}^T$, where $u^*_{k,\eta^\perp_{ij}}$ is defined as follows

$$u^*_{k,\eta^\perp_{ij}} = \begin{cases} \dfrac{q_{k,\eta^\perp_{ij},i}}{h_{k,i}} & \text{if } (\Phi_{\eta_{ij}})_{[2k-1]} > 0 \\ \dfrac{q_{k,\eta^\perp_{ij},j}}{h_{k,j}} & \text{otherwise} \end{cases} \qquad k = 1,2.$$

Let us remark that $\Phi^-_{\eta^\perp_{ij}}$ is the numerical flux associated to the 3-rd and 6-th equations of system (3) where, again, the term $Q_{\eta^\perp_{ij}}$ has been neglected. Its derivation has been done following the main ideas of the HLLC method for the shallow water system introduced in [24] as $q_{k,\eta^\perp_{ij}}$, $k = 1,2$ can be seen as a passive scalar that is convected by the flow.

4. Finally, the global numerical flux is defined by $\mathcal{F}^{IR-ROE^-}_{ij} = T^{-1}_{\eta_{ij}}F^-_{ij}$, where $F^-_{ij} = \begin{bmatrix} (\Phi^-_{\eta_{ij}})_{[1]} & (\Phi^-_{\eta_{ij}})_{[2]} & (\Phi^-_{\eta^\perp_{ij}})_{[1]} & (\Phi^-_{\eta_{ij}})_{[3]} & (\Phi^-_{\eta_{ij}})_{[4]} & (\Phi^-_{\eta^\perp_{ij}})_{[2]} \end{bmatrix}$.

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

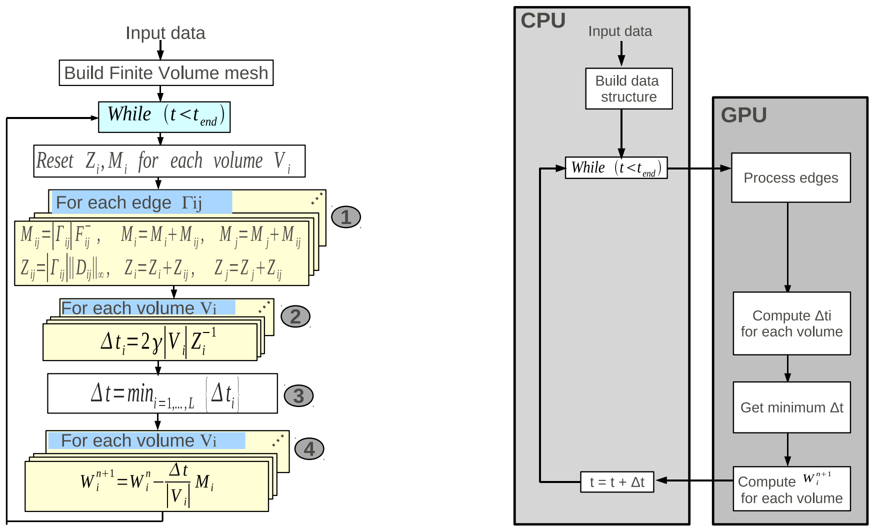
$$\frac{1}{2}\frac{\Delta t}{V_i}\sum_{j\in\mathcal{N}_i}|\Gamma_{ij}|\|D_{ij}\|_\infty \leq \gamma, \quad 1 \leq i \leq L, \text{ with } 0 < \gamma \leq 1. \tag{5}$$

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 [25] can be easily adapted here. Let us also remark that the scheme is path-conservative in the sense introduced by Pares in [26] and [22]. 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 [22] and [27].

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.
Initially, the finite volume mesh is constructed from the input data. Then the time stepping process is repeated until the final simulation time is reached. At the $(n+1)$-th time step, 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:


Input data
Build Finite Volume mesh
While $(t<t_{end})$
Reset Z_i, M_i for each volume V_i
For each edge Γ_{ij}
$M_{ij}=|\Gamma_{ij}|F^-_{ij}$, $M_i=M_i+M_{ij}$, $M_j=M_j+M_{ij}$
$Z_{ij}=|\Gamma_{ij}|\|D_{ij}\|_\infty$, $Z_i=Z_i+Z_{ij}$, $Z_j=Z_j+Z_{ij}$
1
For each volume V_i
$\Delta t_i=2\gamma|V_i|Z_i^{-1}$
2
$\Delta t=min_{i=1,\dots,L}\{\Delta t_i\}$
3
For each volume V_i
$W_i^{n+1}=W_i^n-\frac{\Delta t}{|V_i|}M_i$
4
CPU
Input data
Build data structure
While $(t<t_{end})$
t = t + Δt
GPU
Process edges
Compute Δti for each volume
Get minimum Δt
Compute W_i^{n+1} for each volume

(a) Parallelism sources (b) CUDA implementation

Figure 1: Main calculation phases

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 ($i, j \in \{1, \dots, L\}$):

   a) The computation of the contribution $M_{ij} = \mid \Gamma_{ij} \mid \mathcal{F}_{ij}^{IR-ROE-}$ (a 6×1 vector) to the sums M_i and M_j associated to V_i and V_j (see Eq. (4)).

   b) The computation of the contribution $Z_{ij} = \mid \Gamma_{ij} \mid \| D_{ij}^n \|_\infty$ (a scalar) to the sums Z_i and Z_j associated to V_i and V_j (see Equation (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 [28]. Both programs have been implemented in C++ using double precision and the Eigen library [29] for operating with matrices.

All the programs were executed on a Core i7 920 with 4 GB RAM. Table 1 shows the execution times in seconds. We also show in parenthesis the speedup obtained with the IR-Roe method with respect to the equivalent implementation of the classical Roe method.

As it can be seen, the IR-Roe method clearly outperforms the classical Roe method in all cases, obtaining a speedup of 10 for big meshes. Moreover, this scheme is more suitable to be ported to CUDA-enabled GPUs because

	Classical Roe		IR-Roe			
Volumes	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.

it does not need to use double precision floating point arithmetic and the computational demand of each GPU thread is lower.

5. CUDA Implementation of the IR-Roe method

We describe the CUDA implementation of the algorithm exposed in Section 3. It is a variant of the implementation described in [13], Section 7.3.

The general steps of the parallel implementation are depicted in Figure 1b. Each processing 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 (see [13] for more details):

- **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 volume. 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 contributes to a particular volume 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 the 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 [30].

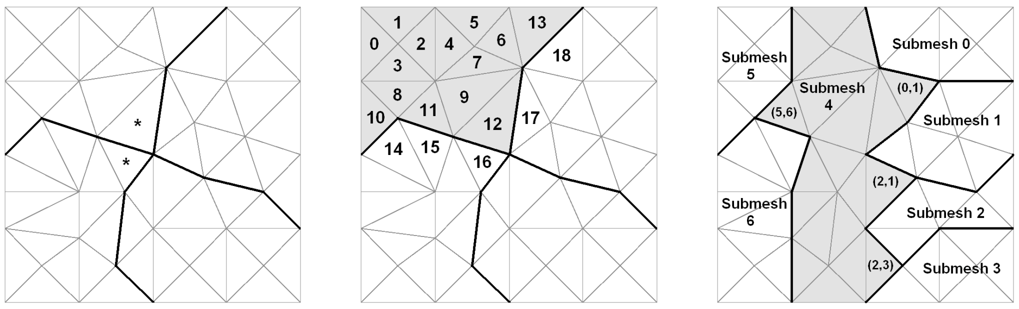
- **Compute W_i for each volume**: In this step, each thread represents a volume and updates the state W_i of the volume V_i. The first 3 elements of M_i are obtained by summing the three 3×1 vectors stored in the positions corresponding to the volume V_i in accumulators 1-1, 2-1 and 3-1, while the last 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 [21] 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 volumes that must be sent to two different MPI processes (i.e. submeshes) in each iteration (**type 1**), and those that do not have volumes 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 volumes 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 volumes, explained in Section 5, but now the arrays of volumes are divided into three blocks:



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

Figure 2: Submeshes.

1. Firstly, the volumes of the submesh that are not communication volumes are stored (a communication volume is a volume that has an adjacent edge to another submesh).
2. Secondly, the communication volumes of the submesh are stored in the array. In turn, these volumes are divided into groups of volumes that are adjacent to a particular submesh.
3. Finally, the communication volumes of another submeshes that are adjacent to our submesh are stored. In turn, these volumes are divided into groups of volumes that belong to a particular submesh.

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

6.2. Creation of Data in Submeshes of Type 1

The creation of the data in submeshes of type 1 is more complicated. For this kind of submeshes, we must arrange the communication volumes of the submesh so that all the communication volumes that are adjacent to a particular submesh appear consecutively in the array. For example, in Figure 2b, note that volumes 10-12 are sent to the lower submesh, while volumes 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 volumes for each adjacent submesh. For example, in Figure 2b we would have two lists: [10, 11, 12] and [12, 13].

Now, for each communication volume of the submesh that must be sent to two MPI processes, we build a pair (p_1, p_2), meaning that the volume 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 volumes that are adjacent to submesh 0, we put the volume shared with submesh 1 at the end.
2. In the list storing the volumes that are adjacent to submesh 1, we put the volume shared with submesh 0 at the start, and the volume 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 volumes adequately to get the definitive block of communication volumes.

Note that a submesh must know the ordering of the communication volumes that receives from another submesh. Therefore, at this point all submeshes must send this information to their adjacent submeshes.

Note also that this algorithm does not work when a submesh has two volumes that must be sent to the same submeshes. This is reflected by a duplicated pair in the sequence of pairs, but we can always perform a domain decomposition where this does not occur. It neither works when a submesh is formed by a single volume, but this will never happen in a real problem.

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 3-5 we send to each adjacent submesh the communication volumes that are adjacent to it. Then, in lines 6-8 we receive from the same submeshes their communication volumes that are adjacent to our submesh. We have used bufferized MPI send operations with a given buffer to avoid deadlocks with big meshes. Lines 9-10 copy the received communication volumes to GPU memory. In line 14 a MPI reduction is performed to obtain the global minimum Δt in all the MPI processes. Lines 16-17 copy the new states of the communication volumes of our submesh from device to host.

Algorithm 2 shows the general steps of the overlapping implementation. Lines 3-5 (the reception of the communication volumes from the adjacent submeshes) overlap with lines 6-7 (the copy of the new states of our communication volumes from device to host). Then, lines 8-10 (the sending to each adjacent submesh of the communication volumes that are adjacent to it)

Algorithm 1 Non-overlapping multi-GPU algorithm

```
1: $n \leftarrow$ number of adjacent submeshes
2: while ($t < t_{end}$) do
3:     for $i = 1$ to $n$ do
4:         Send communication volumes to adjacent submesh $i$
5:     end for
6:     for $i = 1$ to $n$ do
7:         Receive communication volumes from adjacent submesh $i$
8:     end for
9:     CudaMemcpy(Layer 1 of comm. volumes from host to device)
10:    CudaMemcpy(Layer 2 of comm. volumes from host to device)
11:    processEdges<<<grid, block>>>(...)
12:    computeDeltaTVolumes<<<grid, block>>>(...)
13:    $\Delta t \leftarrow$ getMinimumDeltaT(...)
14:    MPI_Allreduce($\Delta t$, min $\Delta t$, ...)
15:    computeVolumeStates<<<grid, block>>>(...)
16:    CudaMemcpy(Layer 1 of comm. volumes from device to host)
17:    CudaMemcpy(Layer 2 of comm. volumes from device to host)
18:    $t \leftarrow t + \min \Delta t$
19: end while
```

overlap with line 11 (the processing of the non-communication edges, since these edges do not need external data to be processed). In line 12 we wait for the communication volumes of the adjacent submeshes to arrive. Once they have arrived, in lines 13-14 we copy them to GPU memory. In line 15 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.

Algorithm 2 Overlapping multi-GPU algorithm

```
1: $n \leftarrow$ number of adjacent submeshes
2: while ($t < t_{end}$) do
3:     for $i = 1$ to $n$ do
4:         Receive comm. volumes from adjacent submesh $i$ (Non-blocking)
5:     end for
6:     CudaMemcpy(Layer 1 of comm. volumes from device to host)
7:     CudaMemcpy(Layer 2 of comm. volumes from device to host)
8:     for $i = 1$ to $n$ do
9:         Send comm. volumes to adjacent submesh $i$ (Non-blocking)
10:    end for
11:    processEdges<<<grid, block>>>(Non-communication edges)
12:    MPI_Waitall (Comm. volumes from adjacent submeshes)
13:    CudaMemcpy(Layer 1 of comm. volumes from host to device)
14:    CudaMemcpy(Layer 2 of comm. volumes from host to device)
15:    processEdges<<<grid, block>>>(Communication edges)
16:    computeDeltaTVolumes<<<grid, block>>>(...)
17:    $\Delta t \leftarrow$ getMinimumDeltaT(...)
18:    MPI_Allreduce($\Delta t$, $\min \Delta t$, ...)
19:    computeVolumeStates<<<grid, block>>>(...)
20:    $t \leftarrow t + \min \Delta t$
21: end while
```

We have used the Chaco software [31] to divide a mesh into equally sized submeshes, the OpenMPI implementation [32] 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 two Tesla C2050 and two GeForce GTX 570. Since the GTX 570 card provides better performance for our programs than the Tesla, it is suitable to use the four cards to measure strong scalability taking the Tesla as the reference card. Table 2 shows the execution times in seconds for all the meshes and number of GPUs. Figure 3a shows graphically the

Volumes	Tesla C2050	2 Tesla C2050		2 Tesla + 2 GTX 570	
		Non-Overlap	Overlap	Non-Overlap	Overlap
4016	0.0090	0.012	0.014	0.011	0.015
16040	0.045	0.038	0.040	0.027	0.031
64052	0.29	0.19	0.19	0.12	0.12
256576	2.10	1.18	1.16	0.69	0.63
1001898	15.63	8.40	7.98	4.60	4.08
2000608	45.37	23.34	23.00	12.08	11.66
3000948	82.84	43.52	41.83	23.34	21.23

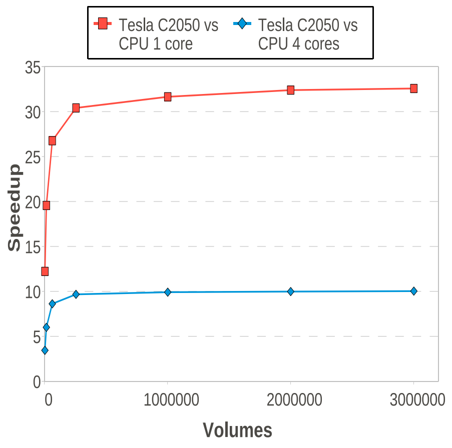
Table 2: GPU execution times in seconds for the IR-Roe method.

speedups obtained with the single GPU program executed on a Tesla C2050 with respect to the CPU versions of the IR-Roe method used in Section 4. Figure 3b shows the speedups obtained with the multi-GPU implementations with respect to one Tesla.

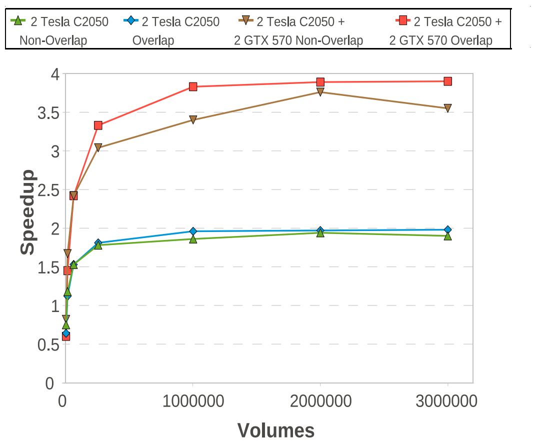
As it can be seen, using a Tesla C2050, for big meshes we have reached a speedup of 32 and 10 with respect to monocore and quadcore CPU versions, respectively. As expected, the overlapping multi-GPU implementation outperforms the non-overlapping version, and the weak and strong scaling reached by the overlapping version are close to perfect 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 volume 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 dis-



(a) Tesla C2050 speedup with respect to serial and quadcore CPU versions



(b) Multi-GPU speedup with respect to one Tesla C2050

Figure 3: Speedups obtained for one and several GPUs.

tributed implementation of this scheme that works on triangular meshes has been implemented using MPI and CUDA. Numerical experiments carried out on a GPU cluster have shown the efficiency of this solver, obtaining weak and strong scaling close to perfect for up to four GPUs by overlapping MPI communications with CPU-GPU memory transfers and GPU computation.

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

Acknowledgements

The authors acknowledges 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 withaPthreads-CUDA implementation onsmall-footprint multi-GPU platforms, The Journal of Supercomputing (2010) 1–27.

[15] 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.

[16] D. Komatitschand, 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.

[17] 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, pp. 47–.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

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

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

[27] 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.

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

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

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

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

[32] 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.