Programming CUDA-based GPUs to simulate two-layer shallow water flows

Marc de la Asunción[1], José M. Mantas[1], and Manuel J. Castro[2]

[1] Dpto. Lenguajes y Sistemas Informáticos, Universidad de Granada
[2] Dpto. Análisis Matemático, Universidad de Málaga

Abstract. The two-layer shallow water system is used as the numerical model to simulate several phenomena related to geophysical flows such as the steady exchange of two different water flows, as occurs in the Strait of Gibraltar, or the tsunamis generated by underwater landslides. The numerical solution of this model for realistic domains imposes great demands of computing power and modern Graphics Processing Units (GPUs) have demonstrated to be a powerful accelerator for this kind of computationally intensive simulations. This work describes an accelerated implementation of a first order well-balanced finite volume scheme for 2D two-layer shallow water systems using GPUs supporting the CUDA (Compute Unified Device Architecture) programming model and double precision arithmetic. This implementation uses the CUDA framewok to exploit efficiently the potential fine-grain data parallelism of the numerical algorithm. Two versions of the GPU solver are implemented and studied: one using both single and double precision, and another using only double precision. Numerical experiments show the efficiency of this CUDA solver on several GPUs and a comparison with an efficient multicore CPU implementation of the solver is also reported.

1 Introduction

The two-layer shallow water system of partial differential equations governs the flow of two superposed shallow layers of immiscible fluids with different constant densities. This mathematical model is used as the numerical model to simulate several phenomena related to stratified geophysical flows such as the steady exchange of two different water flows, as occurs in the Strait of Gibraltar [4], or the tsunamis generated by underwater landslides [14]. The numerical resolution of two-layer or multilayer shallow water systems has been object of an intense research during the last years: see for instance [1–4, 14].

The numerical solution of these equations in realistic applications, where big domains are simulated in space and time, is computationally very expensive. This fact and the degree of parallelism which these numerical schemes exhibit suggest the design of parallel versions of the schemes for parallel machines in order to solve and analyze these problems in reasonable execution times. In this paper, we tackle the acceleration of a finite volume numerical scheme to solve two-layer shallow water systems. This scheme has been parallelized and optimized by combining a distributed implementation which runs on a PC cluster [3] with the use of SSE-optimized routines [5]. However, despite of the important performance improvements, a greater reduction of the runtimes is necessary.

A cost effective way of obtaining a substantially higher performance in these applications consists in using the modern Graphics Processor Units (GPUs). The use of these devices to accelerate computationally intensive tasks is growing in popularity among the scientific and engineering community [12, 15]. Modern GPUs present a massively parallel architecture which includes hundreds of processing units optimized for performing floating point operations and multithreaded execution. These architectures make it possible to obtain performances that are orders of magnitude faster than a standard CPU at a very affordable price.

There are previous proposals to port finite volume one-layer shallow water solvers to GPUs by using a graphics-specific programming language [10, 11]. These solvers obtain considerable speedups to simulate one-layer shallow water systems but their graphics-based design is not easy to understand and maintain.

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

A CUDA solver for one-layer systems based on the finite volume scheme presented in [3] is described in [6]. This one-layer shallow water CUDA solver obtains a good exploitation of the massively parallel architecture of several NVIDIA GPUs. In this work, we extend the proposal presented in [6] for the case of two-layer shallow water systems and we study its performance. From the computational point of view, the numerical solution of the two-layer system presents two main problems with respect to the one-layer case: the need of using double precision arithmetic for some calculations of the scheme and the need of managing a higher volume of data to perform the basic calculations. Our goal is to exploit efficiently the GPUs supporting CUDA and double precision arithmetic in order to accelerate notably the numerical solution of two-layer shallow water systems.

This paper is organized as follows: the next section describes the underlying mathematical model, the two-layer shallow water system, and the finite-volume numerical scheme which has been ported to GPU. A description of the data parallelism of the numerical scheme and its CUDA implementation are presented in Section 4. Section 5 shows and analyzes the performance results obtained when the CUDA solver is applied to several test problems using two different NVIDIA GPUs supporting double precision. Finally, Section 6 summarizes the main conclusions and presents the lines for further work.

2 Mathematical model and Numerical Scheme

The two-layer shallow water system is a system of conservation laws and non-conservative products with source terms which models the flow of two homogeneous fluid shallow layers with different densities that occupy a bounded domain

$D \subset \mathbb{R}^2$ under the influence of a gravitational acceleration g. The system has the following form:

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

being

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

$$B_1(W)=\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & -gh_1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ -rgh_2 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}, \qquad B_2(W)=\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & -gh_1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ -rgh_2 & 0 & 0 & 0 & 0 & 0\end{pmatrix},$$

$$S_1(W)=\begin{pmatrix} 0\\ gh_1\\ 0\\ 0\\ gh_2\\ 0\end{pmatrix}, \qquad S_2(W)=\begin{pmatrix} 0\\ 0\\ gh_1\\ 0\\ 0\\ gh_2\end{pmatrix}$$

where $h_i(x,y,t) \in \mathbb{R}$ denotes the thickness of the water layer i at point (x,y) at time t, $H(x,y) \in \mathbb{R}$ is the depth function measured from a fixed level of reference and $r=\rho_1/\rho_2$ is the ratio of the constant densities of the layers ($\rho_1<\rho_2$), which in realistic oceanographical applications is close to 1 (see Fig. 1). Finally, $q_i(x,y,t)=(q_{i,x}(x,y,t),\, q_{i,y}(x,y,t)) \in \mathbb{R}^2$ is the mass-flow of the water layer i at point (x,y) at time t .

To discretize System (1), the computational domain D is divided into L cells or finite volumes $V_i \subset \mathbb{R}^2$, which are assumed to be quadrangles. Given a finite

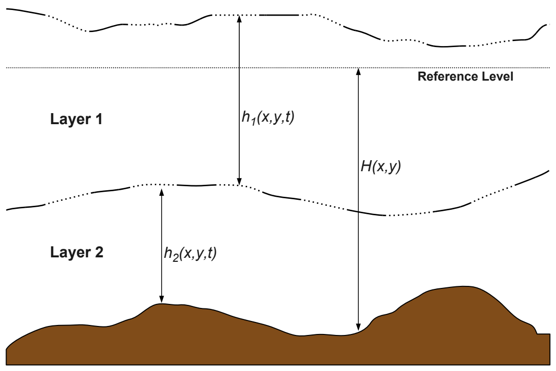
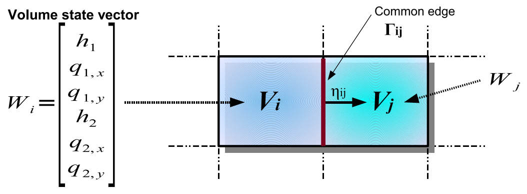


Fig. 1: Two-layer sketch.

volume V_i, $N_i \in \mathbb{R}^2$ is the centre of V_i, $\aleph_i$ is the set of indexes j such that V_j is a neighbour of V_i; $\varGamma_{ij}$ is the common edge of two neighbouring cells V_i and V_j, and $|\varGamma_{ij}|$ is its length; $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unit vector which is normal to the edge $\varGamma_{ij}$ and points towards the cell V_j [3] (see Fig. 2).



Volume state vector
$W_i = \begin{bmatrix} h_1 \\ q_{1,x} \\ q_{1,y} \\ h_2 \\ q_{2,x} \\ q_{2,y} \end{bmatrix}$
Common edge
Γij
Vi
ηij
Vj
Wj

Fig. 2: Finite volumes

Assume that the approximations at time t^n, W_i^n, have already been calculated. To advance in time, with $\varDelta t^n$ being the time step, the following numerical scheme is applied (see [3] for more details):

$$W_i^{n+1} = W_i^n - \frac{\varDelta t^n}{|V_i|} \sum_{j \in \aleph_i} |\varGamma_{ij}| \, F_{ij}^- \tag{2}$$

being

$$F_{ij}^- = \frac{1}{2} \mathcal{K}_{ij} \cdot (I - \operatorname{sgn}(\mathcal{D}_{ij})) \cdot \mathcal{K}_{ij}^{-1} \cdot \left(A_{ij}(W_j^n - W_i^n) - S_{ij}(H_j - H_i) \right),$$

where $|V_i|$ is the area of V_i , $H_l = H(N_l)$ with $l = 1, \ldots, L$, $A_{ij} \in \mathbb{R}^{6\times6}$ and $S_{ij} \in \mathbb{R}^6$ depends on W_i^n and W_j^n, $\mathcal{D}_{ij}$ is a diagonal matrix whose coefficients are the eigenvalues of A_{ij}, and the columns of $K_{ij} \in \mathbb{R}^{6\times 6}$ are the associated eigenvectors.

To compute the n-th time step, the following condition can be used:

$$\Delta t^n = \min_{i=1,\ldots,L} \left\{ \left[\frac{\sum_{j\in\aleph_i} \mid \Gamma_{ij} \mid \parallel \mathcal{D}_{ij} \parallel_\infty}{2\gamma \mid V_i \mid} \right]^{-1} \right\} \tag{3}$$

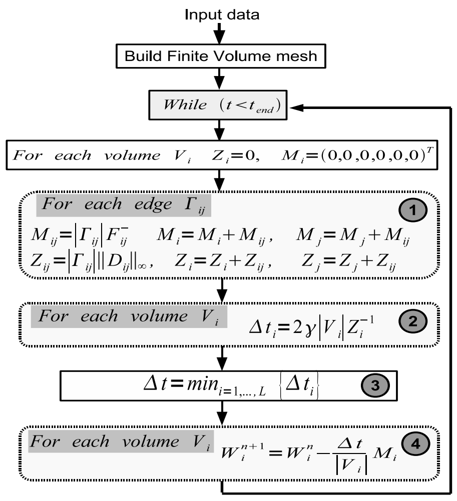
where γ, $0 < \gamma \leq 1$, is the CFL (Courant-Friedrichs-Lewy) parameter.

3 CUDA Implementation

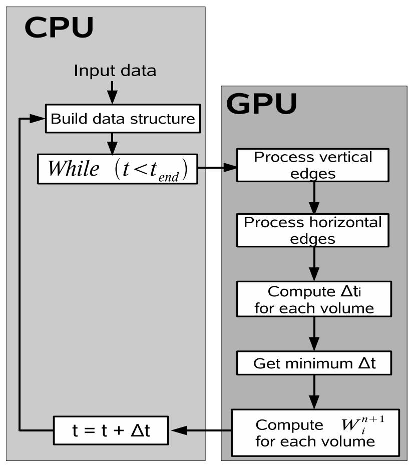
In this section we describe the potential data parallelism of the numerical scheme and its implementation in CUDA.

3.1 Parallelism sources

Figure 3a shows a graphical description of the main sources of parallelism obtained from the numerical scheme. The main calculation phases, identified with circled numbers, presents a high degree of parallelism because the computation performed at each edge or volume is independent with respect to that performed at other edges or volumes.


Input data
Build Finite Volume mesh
While $(t<t_{end})$
For each volume V_i $Z_i=0$, $M_i=(0,0,0,0,0,0)^T$
For each edge Γ_{ij}
1
$M_{ij}=|\Gamma_{ij}|F_{ij}^-$ $M_i=M_i+M_{ij}$, $M_j=M_j+M_{ij}$
$Z_{ij}=|\Gamma_{ij}|\|D_{ij}\|_\infty$, $Z_i=Z_i+Z_{ij}$, $Z_j=Z_j+Z_{ij}$
For each volume V_i
2
$\Delta t_i=2\gamma|V_i|Z_i^{-1}$
$\Delta t=min_{i=1,\ldots,L}\{\Delta t_i\}$
3
For each volume V_i
4
$W_i^{n+1}=W_i^n-\frac{\Delta t}{|V_i|}M_i$

(a) Parallelism sources of the numerical scheme



(b) General steps of the parallel algorithm implemented in CUDA

Fig. 3: Parallel algorithm.

When the finite volume mesh has been constructed, the time stepping process is repeated until the final simulation time is reached:

1. **Edge-based calculations**: Two calculations must be performed for each edge Γ_{ij} communicating two cells V_i and V_j $(i, j \in \{1, \ldots, L\})$:
   a) Vector $M_{ij} = |\,\Gamma_{ij}\,|\, F_{ij}^- \in \mathbb{R}^6$ must be computed as the contribution of each edge to the calculation of the new states of its adjacent cells V_i and V_j (see (3)). This contribution can be computed independently for each edge and must be added to the partial sums M_i and M_j associated to V_i and V_j, respectively.
   b) The value $Z_{ij} = |\,\Gamma_{ij}\,|\,\|\,\mathcal{D}_{ij}\,\|_\infty$ must be computed as the contribution of each edge to the calculation of the local Δt values of its adjacent cells V_i and V_j (see (2)). This contribution can be computed independently for each edge and must be added to the partial sums Z_i and Z_j associated to V_i and V_j, respectively.
2. **Computation of the local Δt_i for each volume**: For each volume V_i, the local Δt_i is obtained as follows (see (3)): $\Delta t_i = 2\gamma\,|V_i|\,Z_i^{-1}$. In the same way, the computation for each volume can be performed in parallel.
3. **Computation of Δt**: The minimum of all the local Δt_i values previously computed for each volume is obtained. This minimum Δt represents the next time step which will be applied in the simulation.
4. **Computation of W_i^{n+1}**: The $(n+1)$-th state of each volume (W_i^{n+1}) is calculated from the n-th state and the data computed in previous phases, in the following way (see (2)): $W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|}\,M_i$. This phase can also be performed in parallel (see Fig. 3a).

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

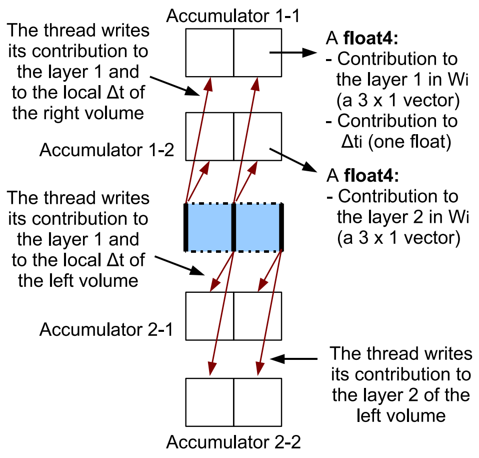
4 Algorithmic details of the CUDA version

In this section we describe the parallel algorithm we have developed and its implementation in CUDA. It is an extension of the algorithm described in [6] to simulate two-layer shallow water systems. We consider problems consisting in a bidimensional regular finite volume mesh. The general steps of the parallel algorithm are depicted in Fig. 3b. Each processing step executed on the GPU is assigned to a CUDA kernel. A kernel is a function executed on the GPU by many threads which are organized forming a grid of thread blocks that run logically in parallel (see [7] for more details). Next, we describe in detail each step:

– **Build data structure**: In this step, the data structure that will be used on the GPU is built. For each volume, we store its initial state (h_1, $q_{1,x}$, $q_{1,y}$, h_2, $q_{2,x}$ and $q_{2,y}$) and its depth H. We define two arrays of `float4` elements, where each element represents a volume. The first array contains h_1, $q_{1,x}$, $q_{1,y}$ and H, while the second array contains h_2, $q_{2,x}$ and $q_{2,y}$. Both arrays are stored as 2D textures.
  The area of the volumes and the length of the vertical and horizontal edges are precalculated and passed to the CUDA kernels that need them.
  We can know at runtime if an edge or volume is frontier and the value of the normal $\boldsymbol{\eta}_{ij}$ of an edge by checking the position of the thread in the grid.

– **Process vertical edges** and **process horizontal edges**: As in [6], we divide the edge processing into vertical and horizontal edge processing. For vertical edges, $\eta_{ij,y} = 0$ and therefore all the operations where this term takes part can be discarded. Similarly, for horizontal edges, $\eta_{ij,x} = 0$ and all the operations where this term takes part can be avoided.
In vertical and horizontal edge processing, each thread represents a vertical and horizontal edge, respectively, and computes the contribution of the edge to their adjacent volumes as described in section 3.1.
The edges (i.e. threads) synchronize each other when contributing to a particular volume by means of four accumulators (in [6] we used two accumulators for one-layer systems), each one being an array of `float4` elements. The size of each accumulator is the number of volumes. Let us call the accumulators 1-1, 1-2, 2-1 and 2-2. Each element of accumulators 1-1 and 2-1 stores the contributions of the edges to the layer 1 of W_i (the first 3 elements of M_i) and to the local Δt of the volume (a `float` value Z_i), while each element of accumulators 2-1 and 2-2 stores the contributions of the edges to the layer 2 of W_i (the last 3 elements of M_i). Then, in the processing of vertical edges:
  - Each vertical edge writes in the accumulator 1-1 the contribution to the layer 1 and to the local Δt of its right volume, and writes in the accumulator 1-2 the contribution to the layer 2 of its right volume.
  - Each vertical edge writes in the accumulator 2-1 the contribution to the layer 1 and to the local Δt of its left volume, and writes in the accumulator 2-2 the contribution to the layer 2 of its left volume.

Next, the processing of horizontal edges is performed in an analogous way, but with the difference that the contribution is added to the accumulators instead of only writing it. Figure 4 shows this process graphically.


Accumulator 1-1
The thread writes its contribution to the layer 1 and to the local Δt of the right volume
A **float4:**
- Contribution to the layer 1 in Wi (a 3 x 1 vector)
- Contribution to Δti (one float)
Accumulator 1-2
A **float4:**
- Contribution to the layer 2 in Wi (a 3 x 1 vector)
The thread writes its contribution to the layer 1 and to the local Δt of the left volume
Accumulator 2-1
The thread writes its contribution to the layer 2 of the left volume
Accumulator 2-2
(a) Vertical edge processing

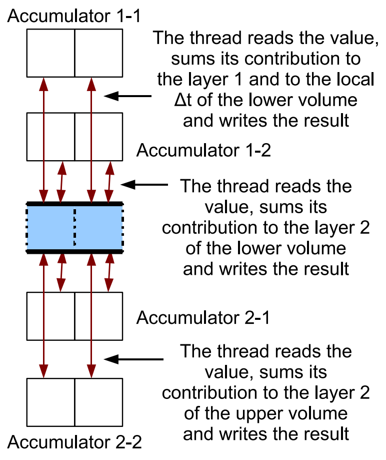
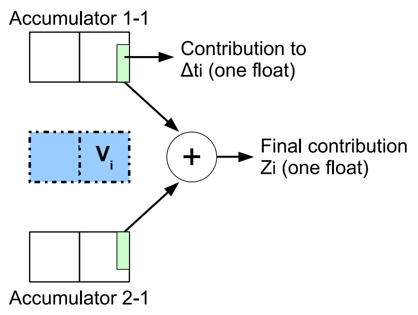
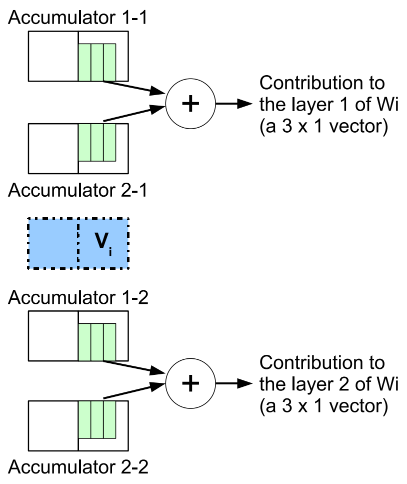

Accumulator 1-1
The thread reads the value, sums its contribution to the layer 1 and to the local Δt of the lower volume and writes the result
Accumulator 1-2
The thread reads the value, sums its contribution to the layer 2 of the lower volume and writes the result
Accumulator 2-1
The thread reads the value, sums its contribution to the layer 2 of the upper volume and writes the result
Accumulator 2-2
(b) Horizontal edge processing

Fig. 4: Computing the sum of the contributions of the edges of each volume.



(a) Contribution to Δt_i



(b) Contribution to W_i

Fig. 5: Computation of the final contribution of the edges for each volume.

- **Compute Δt_i for each volume**: In this step, each thread represents a volume and computes the local Δt_i of the volume V_i as described in section 3.1. The final Z_i value is obtained by summing the two float values stored in the positions corresponding to the volume V_i in accumulators 1-1 and 2-1 (see Fig. 5a).
- **Get minimum Δt**: This step finds the minimum of the local Δt_i of the volumes by applying a reduction algorithm on the GPU. The reduction algorithm applied is the kernel 7 (the most optimized one) of the reduction sample included in the CUDA Software Development Kit [8].
- **Compute W_i for each volume**: In this step, each thread represents a volume and updates the state W_i of the volume V_i as described in section 3.1. The final M_i value is obtained as follows: the first 3 elements of M_i (the contribution to layer 1) are obtained by summing the two 3×1 vectors stored in the positions corresponding to the volume V_i in accumulators 1-1 and 2-1, while the last 3 elements of M_i (the contribution to layer 2) are obtained by summing the two 3×1 vectors stored in the positions corresponding to the volume V_i in accumulators 1-2 and 2-2 (see Fig. 5b). Since a CUDA kernel can not write directly into textures, the textures are updated by firstly writing the results into temporary arrays and then these arrays are copied to the CUDA arrays bound to the textures.

A version of this CUDA algorithm which uses double precision to perform all the computing phases has also been implemented. The volume data is stored in three arrays of `double2` elements (which contain the state of the volumes) and one array of `double` elements (the depth H). We use six accumulators of `double2` elements (for storing the contributions to W_i) and two accumulators of `double` elements (for storing the contributions to the local Δt of each volume).

5 Experimental Results

We consider an internal circular dambreak problem in the $[-5,5] \times [-5,5]$ rectangular domain in order to compare the performance of our implementations. The depth function is given by $H(x,y) = 2$ 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

$$h_1(x,y) = \begin{cases} 1.8 & \text{if } \sqrt{x^2+y^2} > 4 \\ 0.2 & \text{otherwise} \end{cases}, \qquad h_2(x,y) = 2 - h_1(x,y)$$

The numerical scheme is run for several regular bidimensional finite volume meshes with different number of volumes (see Table 1). Simulation is carried out in the time interval $[0,1]$. 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.

To perform the experiments, several programs have been implemented:

– *A serial CPU version of the CUDA algorithm.* This version has been implemented in C++ and uses the Eigen library [9] for operating with matrices. We have used the `double` data type in this implementation.
– *A quadcore CPU version of the CUDA algorithm.* This is a parallelization of the aforementioned serial CPU version which uses OpenMP [13].
– *A mixed precision CUDA implementation (CUSDP).* In this GPU version, the eigenvalues and eigenvectors of the A_{ij} matrix (see Sect. 2) are computed using double precision to avoid numerical instability problems, but the rest of operations are performed in single precision.
– *A full double precision CUDA implementation (CUDP).*

All the programs were executed on a Core i7 920 with 4 GB RAM. Graphics cards used were a GeForce GTX 280 and a GeForce GTX 480. Figure 7 shows the evolution of the fluid. Table 1 shows the execution times in seconds for all the meshes and programs. As can be seen, the number of volumes and the execution times scale with a different factor because the number of time steps required for the same time interval also augments when the number of cells is increased (see (3)). Using a GTX 480, for big meshes, CUSDP achieves a speedup of 62 with respect to the monocore CPU version, while CUDP reaches a speedup of 38. As expected, the OpenMP version only reaches a speedup less than four in all meshes. CUDP has been about 38 % slower than CUSDP for big meshes in the GTX 480 card, and 24 % slower in the GTX 280 card.

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

Table 1: Execution times in seconds for all the meshes and programs.

Mesh size	CPU	CPU	GTX 280		GTX 480	
$L = L_x \times L_y$	1 core	4 cores	CUSDP	CUDP	CUSDP	CUDP
100×100	7.54	2.10	0.48	0.80	0.37	0.53
200×200	59.07	15.84	3.15	4.38	1.42	2.17
400×400	454.7	121.0	21.92	29.12	8.04	13.01
800×800	3501.9	918.7	163.0	216.1	57.78	94.57
1600×1600	28176.7	7439.4	1262.7	1678.0	453.5	735.6
2000×2000	54927.8	14516.6	2499.2	3281.0	879.7	1433.6

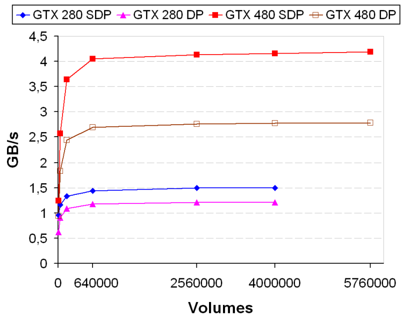
Table 2: Mean values of the percentages of the execution time and GPU FLOPS for all the computing steps.

	% Execution time				% GPU
Computing step	1 core	4 cores	CUSDP	CUDP	FLOPS
Process vertical edges	49.6	48.2	49.5	50.0	49.5
Process horizontal edges	49.8	48.6	49.4	48.5	49.9
Compute Δt_i	0.2	1.1	0.3	0.3	0.1
Get minimum Δt		0.4	0.1	0.2	0.0
Compute W_i^{n+1}	0.4	1.7	0.7	1.0	0.5

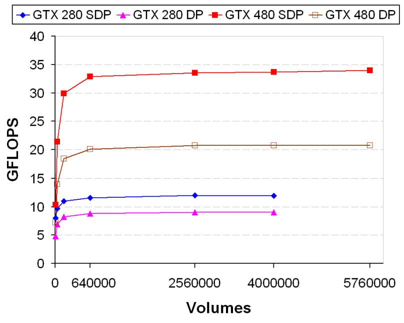
Table 2 shows the mean values of the percentages of the execution time and GPU FLOPS for all the computing steps and implementations. Clearly, almost all the the execution time is spent in the edge processing steps.

Figure 6 shows graphically the GB/s and GFLOPS obtained in the CUDA implementations with both graphics cards. In the GTX 480 card, CUSDP achieves 4.2 GB/s and 34 GFLOPS for big meshes. Theoretical maximums are: for the GTX 480, 177.4 GB/s, and 1.35 TFLOPS in single precision, or 168 GFLOPS in double precision; for the GTX 280, 141.7 GB/s, and 933 GFLOPS in single precision, or 78 GFLOPS in double precision.

As can be seen, the speedup, GB/s and GFLOPS reached with the CUSDP program are notably worse than those obtained in [6] with the single precision CUDA implementation for one-layer systems. This is mainly due to two reasons. Firstly, since double precision has been used to compute the eigenvalues and eigenvectors, the efficiency is reduced because the double precision speed is 1/8 of the single precision speed in GeForce cards with GT200 and GF100 architectures. Secondly, since the register usage and the complexity of the code executed by each thread is higher in this implementation, the CUDA compiler has to store some data into local memory, which also increases the execution time.

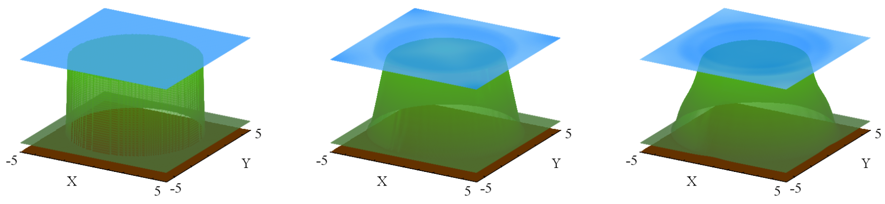


(a) GB/s



(b) GFLOPS

Fig. 6: GB/s and GFLOPS obtained with the CUDA implementations in all meshes with both graphics cards.



(a) t = 0.0 (b) t = 2.5 (c) t = 5.0

Fig. 7: Graphical representation of the fluid evolution at different time instants.

We also have compared the numerical solutions obtained in the monocore and the CUDA programs. The L1 norm of the difference between the solutions obtained in CPU and GPU at time $t = 1.0$ for all meshes was calculated. The order of magnitude of the L1 norm using CUSDP vary between 10^{-4} and 10^{-6}, while that of obtained using CUDP vary between 10^{-12} and 10^{-14}, which reflects the different accuracy of the numerical solutions computed on the GPU using both single and double precision, and using only double precision.

6 Conclusions and further work

In this paper we have presented an efficient first order well-balanced finite volume solver for two-layer shallow water systems. The numerical scheme has been parallelized, adapted to the GPU and implemented using the CUDA framework in order to exploit the parallel processing power of GPUs. On the GTX 480 graphics card, the CUDA implementation using both single and double precision has reached 4.2 GB/s and 34 GFLOPS, and has been one order of magnitude faster than a monocore CPU version of the solver for big uniform meshes. It is expected that this results will significantly improve on a NVIDIA Tesla GPU architecture based on Fermi, since this architecture includes more double precision support than the GTX 480 graphics card. The simulations carried out also reveal the different accuracy obtained with the two implementations of the solver, getting better accuracy using double precision than using both single and double precision. As further work, we propose to extend the strategy to enable efficient simulations on irregular and non-structured meshes.

Acknowledgements

J. M. Mantas acknowledges partial support from the DGI-MEC project MTM2008-06349-C03-03. M. de la Asunción and M. J. Castro acknowledge partial support from DGI-MEC project MTM2009-11923.

References

1. Abgrall, R., Karni, S.: Two-layer shallow water system: a relaxation approach. SIAM J. Sci. Comput. 31, 1603–1627 (2009).
2. Audusse, E., Bristeau, M.O.: Finite-volume solvers for a multilayer Saint-Venant system. Int. J. Appl. Math. Comput. Sci. 17, 311–319 (2007).
3. Castro, M. J., García-Rodríguez, J. A., González-Vida J. M., Parés, C.: A parallel 2d finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows. Comput. Meth. Appl. Mech. Eng. 195, 2788–2815 (2006).
4. Castro, M. J., García, J. A., González-Vida J.M., Macías J., Parés C.: Improved FVM for two-layer shallow-water models: Application to the Strait of Gibraltar. Adv. in Eng. Soft. 38, 386–398 (2007).
5. Castro, M. J., García-Rodríguez, J. A., González-Vida, J. M., Parés, C.: Solving shallow-water systems in 2D domains using Finite Volume methods and multimedia SSE instructions. J. Comput. Appl. Math. 221, 16–32 (2008).
6. de la Asunción, M., Mantas, J. M., Castro, M. J.: Simulation of one-layer shallow water systems on multicore and CUDA architectures. J. Supercomput. (DOI: 10.1007/s11227-010-0406-2).
7. NVIDIA: NVIDIA CUDA Programming Guide Version 3.0 (2010). http://developer.nvidia.com/object/cuda_3_0_downloads.html
8. NVIDIA: CUDA home page, http://www.nvidia.com/object/cuda_home_new.html.
9. Eigen 2.0.12., http://eigen.tuxfamily.org.
10. Hagen, T. R., Hjelmervik, J. M., Lie, K.-A., Natvig, J. R., Ofstad Henriksen, M.: Visual simulation of shallow-water waves. Simul. Model. Pract. Theory, 13, 716–726 (2005).
11. Lastra, M., Mantas, J. M., Ureña, C., Castro, M. J., García, J. A.: Simulation of shallow-water systems using graphics processing units. Math. Comput. Simulat. 80, 598–618 (2009).
12. Rumpf, M., Strzodka, R.: Graphics processor units: New prospects for parallel computing, Lecture Notes Comput. Sci. Eng., 51, 89–121 (2006).
13. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable shared memory parallel programing. The MIT Press (2007).

14. Ostapenko, V. V.: Numerical simulation of wave flows caused by a shoreside landslide. J. Appl. Mech. Tech. Phys. 40, 647–654 (1999).
15. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C: GPU Computing. Proceedings of the IEEE 96, 879–899 (2008).