Simulation of Shallow Water systems using GPUs

M. Lastra[1] and J. M. Mantas[1] and C. Ureña [1] and M. J. Castro[2] and J.A García-Rodríguez [3]

Abstract— **We address the speedup of the numerical solution of shallow water systems in 2D domains by using modern Graphics Processing Units. We have considered a first order well-balanced finite volume numerical scheme for 2D shallow water systems. The potential data parallelism of this method has been identified and an efficient implementation of this scheme to solve one-layer shallow-water systems has been derived for GPUs. Numerical experiments which have been performed on several GPUs show the high efficiency of the GPU solver in comparison with a highly optimized implementation of a CPU solver.**

Keywords— **shallow water simulation, General Purpose computation on Graphics Processing Units (GPGPU), high performance scientific computing.**

I. Introduction

OU r goal is to efficiently simulate one layer fluids that can be modeled by using a shallow water system, formulated under the form of a conservation law with source terms. The numerical solution of these models is useful for several applications related to geophysical flows: simulation of rivers, channels, dambreak problems, etc. However, these simulations impose a great demand of computing power due to the dimensions of the domain (space and time). As a consequence, extremely efficient high performance solvers are required to solve and analyze these problems in reasonable execution times. An interesting numerical scheme to simulate shallow water systems and an efficient parallel implementation of this scheme for a PC cluster has been presented in [1]. This parallel implementation of the numerical scheme has been improved by using SSE-optimized software modules in order to accelerate small matrix computations at each processing node of the cluster (see [2]). Although these improvements have made it possible to obtain results in lower computational times, the simulations still require too much runtime despite of using efficiently all the resources of a powerful PC cluster.

Currently, a cost effective emerging architecture exists which is specially indicated to accelerate considerably computationally intensive tasks like the one considered in this paper. Modern Graphics Processing Units (GPUs) are not only used to render 3D graphics but can also be a cost effective way to speedup the numerical solution of several mathematical models in science and engineering (see [6], [11] for a revision of the topic). Modern GPUs offer over 100 processing units optimized for performing massively floating point operations in parallel [9]. As a consequence, for several algorithmic structures, these architectures are able to obtain a substantially higher performance than a powerful CPU.

In [7], a explicit central-upwind scheme is implemented on a NVIDIA GeForce7800 GTX card to simulate the one-layer shallow-water system and a speedup from 15 to 30 is achieved with respect an implementation on an Intel Xeon processor.

We propose an strategy to design an efficient implementation of the numerical scheme presented in [1] on GPUs using OpenGL and Cg [4]. For that, we needed to adapt the calculations and the data domain of the numerical algorithm to the graphics processing pipeline. We have developed a utility library which facilitates the mapping and simplifies the description of the GPU program as sequential composition of data parallel modules (herein called fragment shaders).

In the next section, we describe the structure of the one-layer shallow-water system. Section III introduces the underlying numerical scheme. In Section IV, the data parallelism of the scheme and the main calculation phases are identified. The design of the GPU version of the numerical solver is described in Section V. In Section VI, we show and analyze the results obtained when the solvers are applied to several meshes using several GPUs. Finally, in Section VII we collect the main conclusions of the work and present the lines of further work.

II. Mathematical model: One-layer shallow-water system

The one-layer shallow-system is a system of conservation laws with source terms which models the flow of a shallow layer of homogeneous fluid that occupies a bounded subdomain $D \subset \mathbb{R}^2$ under the influence of a gravitational acceleration g. The system has the following form:

$$\begin{cases} \frac{\partial h}{\partial t} + \frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} = 0 \\ \frac{\partial q_x}{\partial t} + \frac{\partial}{\partial x}\left(\frac{q_x^2}{h} + \frac{g}{2}h^2\right) + \frac{\partial}{\partial y}\left(\frac{q_x q_y}{h}\right) = gh\frac{\partial H}{\partial x} \\ \frac{\partial q_y}{\partial t} + \frac{\partial}{\partial x}\left(\frac{q_x q_y}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q_y^2}{h} + \frac{g}{2}h^2\right) = gh\frac{\partial H}{\partial y} \end{cases} \quad (1)$$

where $h(x, y, t) \in \mathbb{R}$ denotes the thickness of the water layer at point (x, y) at time t, $H(x, y)$ is the depth

[1]Depto. de Lenguajes y Sistemas Informáticos, E.T.S. Ingeniería Informática y Telecomunicaciones, Univ. de Granada. 18071 Granada. e-mails: mlastral@ugr.es, jmmantas@ugr.es, curena@ugr.es

[2]Depto. de Análisis Matemático, Facultad de Ciencias, Univ. de Málaga. 29071. Málaga. e-mail: castro@anamat.cie.uma.es

[3]Dpto. de Matemáticas, Universidad de A Coruña, Campus de Elviña s/n, 15071 A Coruña. Spain. e-mail: jagrodriguez@udc.es

function measured from a fixed level of reference and $q(x,y,t) = (q_x(x,y,t), q_y(x,y,t)) \in \mathbb{R}^2$ is the mass-flow of the water layer at point (x,y) at time t.

Our problem consists of studying the time evolution of $W(x,y,t) = [h(x,y,t), q_x(x,y,t), q_y(x,y,t)]^T$ satisfying the system (1).

III. Numerical scheme

As described in [1], the system (1) is discretized by means of a finite volume scheme. Domain D is divided into M finite volumes or discretization cells which are supposed to be closed polygons. Given a cell $V_i \subset \mathbb{R}^2$, $i = 1, \ldots, M$, with area $\mid V_i \mid$, $N_i \in \mathbb{R}^2$ is the center of V_i, Γ_{ij} is the common edge of two neighbor cells V_i and V_j with length $\mid \Gamma_{ij} \mid$, $\eta_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unit vector which is normal to the edge Γ_{ij} and points towards V_j (see Figure 1).

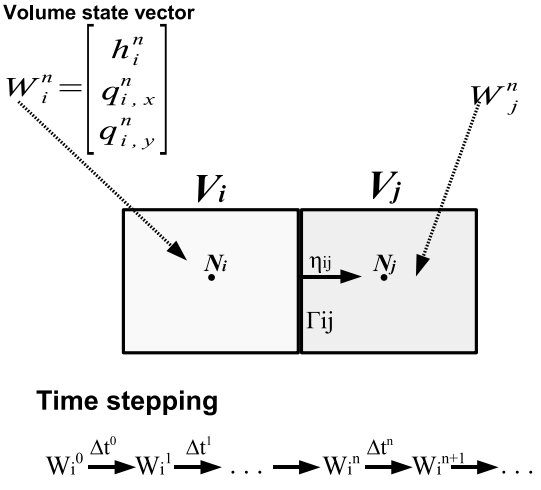

Fig. 1. Finite volumes

The approximations to the cell averages of the exact solution produced by the numerical scheme at time t^n is denoted by $W_i^n = (h_i^n, q_{i,x}^n, q_{i,y}^n)^T$. If we suppose that W_i^n have been already calculated, the approximation W_i^{n+1} is computed using a path-conservative Roe scheme described in [2], [3], [10], as follows:

$$W_i^{n+1} = W_i^n - \frac{\Delta t^n}{\mid V_i \mid} \sum_{j \in \aleph_i} \mid \Gamma_{ij} \mid F_{ij}^n \qquad (2)$$

where $\aleph_i$ is the set of indexes j such that V_j is a neighbor of V_i, Δt^n is the n-th time step ($\Delta t^n = t^{n+1} - t^n$). The term $F_{ij}^n \in \mathbb{R}^3$ is computed as shown below:

$$\begin{aligned} F_{ij}^n &= P_{ij}^n \left[A_{ij}^n (W_j^n - W_i^n) - S_{ij}^n (H_j - H_i) \right], \\ P_{ij}^n &= \frac{1}{2} K_{ij}^n \cdot \left[I - sgn(D_{ij}^n) \right] \cdot (K_{ij}^n)^{-1}, \end{aligned}$$

where $A_{ij}^n \in \mathbb{R}^{3\times 3}$ and $S_{ij}^n \in \mathbb{R}^3$ depends on W_i^n and W_j^n, D_{ij}^n is a diagonal matrix whose coefficients are the eigenvalues of A_{ij}^n and the columns of $K_{ij}^n \in \mathbb{R}^{3\times 3}$ are the associated eigenvectors (see [1]) to obtain more details about the computation of these matrices).

The time step Δt^n is computed to satisfy the usual CFL condition as follows:

$$\Delta t^n = \min_{i=1,\ldots,M} \left\{ \left[\frac{\sum_{j \in \aleph_i} \mid \Gamma_{ij} \mid\mid\mid D_{ij}^n \mid\mid_\infty}{2\gamma \mid V_i \mid} \right]^{-1} \right\} \qquad (3)$$

being γ, $0 < \gamma \leq 1$, the CFL parameter..

The previous numerical scheme is exactly well-balanced for the steady solution corresponding to water at rest. A high order extension of the previous numerical scheme has been presented in [3]. An high order numerical treatment of the wet-dry fronts has been introduced in [5] which ensures the positivity of the water depth at the front as well as well-balanced properties of original first order scheme. Finally, extensions to purely non-conservative hyperbolic systems, like the two-layer shallow-water system have been also performed (see [2], [10]).

IV. Parallelism identification and numerical algorithm

We have designed a data parallel numerical algorithm from the mathematical description of the numerical scheme. Figure 2 shows a graphical description of the parallel numerical algorithm. In this figure, the main calculation phases have been identified with circled numbers and the the main sources of data parallelism have been clearly indicated.

Initially, the finite volume mesh must be constructed from the input data with the appropriate setting of initial and boundary conditions.Then the time stepping process is repeated until the final simulation time is reached. At the $n+1$-th time step, Equation 2 must be evaluated to update the state of each cell. There is of course a dependence of the data that gets computed at the $n+1$-time step with respect to the previous one, but this fact does not impose any restriction to parallelize the computations performed at a time step. In fact, the four main calculation phases of the evaluation present a lot of parallelism and must be completed consecutively:

1. Edge-based calculations: Two calculations must be performed for each edge Γ_{ij} communicating two cells V_i and V_j $(i, j \in 1, \ldots, M)$:

a) Vector $M_{ij} = \mid \Gamma_{ij} \mid F_{ij}^n \in \mathbb{R}^3$ must be computed as contribution of each edge to the sum associated to the corresponding neighbor cells V_i and V_j (see Equation 2) This contribution must be added to the partial sums associated to each cell (M_i and M_j). The computation of this contribution can be computed independently for each edge and it is the most costly calculation in the numerical algorithm because it includes several 3×3 matrix computations (inversion, matrix-matrix product, matrix-vector product, etc.). Moreover, since we only need the data corresponding to the volumes V_i and V_j to compute the contribution for one particular edge Γ_{ij}, this computation presents a high arithmetic intensity and locality.

b) The value $\Delta t_{ij} = \mid \Gamma_{ij} \mid\mid\mid D_{ij}^n \mid\mid_\infty$ must be computed and added to the partial sums associated to each cell (Δt_i and Δt_j) as an intermediate step to compute Δt^n (see Equation 3).

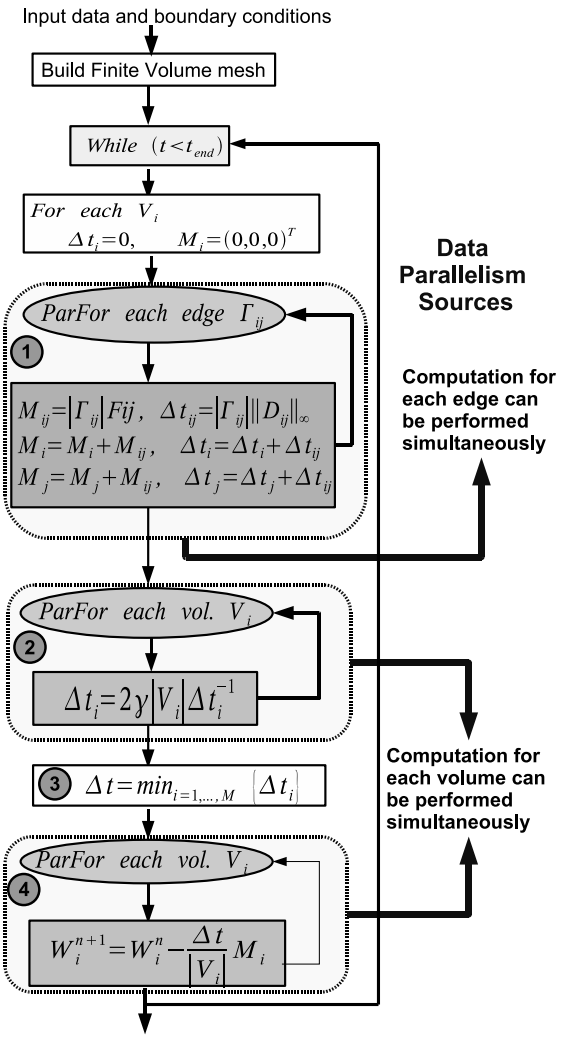

Fig. 2. Main calculation phases in the parallel algorithm

Both calculations for an edge can be computed simultaneously with respect the calculations associated to another edges.

2. Computation of the local Δt for each volume: For each volume V_i, the value of Δt_i is modified to compute the local Δt per volume. In the same way, the computation for each volume can be performed in parallel.

3. Computation of Δt^n: The minimum of all the local Δt values previously obtained for each volume must be computed. This phase can also be parallelized if the minimum is calculated following a recursive decomposition approach [8].

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 at the previous phases. This phase can also be completed in parallel (see Figure 2).

We can make the following remarks from the description of the parallel algorithm: a) the computation steps required by the problem presented in this paper can be classified into two groups: the computation associated to edges and the computation associated to volumes; b) the scheme presents a high arithmetic intensity and the computation exhibits a high degree of locality because the computation for each edge or volume only depends on data from adjacent volumes; c) the scheme exhibits a high degree of potential data parallelism (see Figure 2) because the computation at each edge or volume is independent with respect to the computation performed or associated to the rest of edges or volumes.

V. Design and implementation of the GPU solver

The remarks indicate that this problem seems suitable for being implemented on modern graphics processing units. These processors offer over 100 processing units optimized for performing operations with 4-tuples or 4x4 matrices of floating point numbers (also with smaller tuples and matrices) and floating point operations in general. In the numerical scheme presented, the volume state is represented by a 3-tuple and all the operations involve operations between 3-tuples and 3x3 matrices which makes it even more suited for a GPU based computing platform. The only drawback of using GPUs is the need to adapt the computational process to the graphics processing pipeline and make some mappings between the problem domain and this pipeline.

A. Data storage and arrangement in the GPU

In Computer Graphics most of the data is represented by 3 or 4-tuples to denote points and vectors. Transformations are usually represented by 3x3 or 4x4 matrices. These matrices are multiplied to obtain the composition of different transformations and vectors and points are multiplied by these matrices to apply transformations. Therefore these types of data storage and operations are highly optimized.

There is another type of data which is commonly used: textures. A 2D texture allows the storage of $n \times m$ floating point 4-tuples and are mainly used to store colors representing an image to be applied to a 3D object. In order to map 3D points and their corresponding texture position, texture coordinates are assigned to 3D objects. 3D textures are also available but they are not required in this context.

The aforementioned mechanisms allow the storage and representation of the data required by the numerical solver. In fact, the most important data about volumes and edges must be stored as 2D textures.

A.1 Volume-based information textures

Volumes require the storage of both, the data which remains constant during the computation, and the data related to the current and the next state. The constant data is the following: the depth function (H), the volume area , the information about the orientation of the normal vector associated to each edge and an indication about whether the volume is a ghost volume or not (fictitious cells which are only used to impose the boundary conditions). This data can be packed in a vector of 4 floating point numbers (float4): one floating point value for H, one for the area, one for the normal orientation in-

formation and one to indentify a ghost volume. The penultimate term requires some additional information about how the orientation of the normals has been coded into a floating point value. Basically the above-mentioned floating point number which represents the normal vectors orientations will be treated as an integer value and the orientation of the vectors is obtained by evaluating the value of its four less significant bits. A value of 1 means the normal points towards the exterior of the volume. The mapping between bits and edges is shown in Figure 3 a):

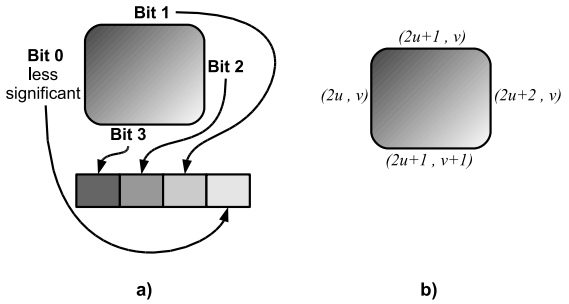

Fig. 3. Edges-bits mapping and edge coordinates for a cell

These 3-tuples are stored in a $n \times m$ texture where $n \times m$ is at least equal to the number of volumes (see Figure 4). It is a rectangular matrix where each position contains a 3-tuple and is associated to a volume. On this texture there would still be memory for an additional floating point value per volume, and thus float4 data type could be used instead of a float3 one.

The volume state data includes 3 floating point values. Each W_i^n vector represents the state of volume i at the n-th time step. This data is stored in another texture where there would also be a space for an additional floating point value per volume.

A.2 Edge-based information texture

Regarding edges, the information to be stored is: the normal vector 2D coordinates, the length and a value that indicates whether the edge is a boundary edge or not. Again a rectangular texture of 3-tuples allows the presentation of this information. The third number will be set to be positive on edges which are at the frontier of the volume grid.

The edge information textures have the same rows as the volume information texture, but more columns because the number of edges is higher. At each row, for each volume, from left to right, the information of the left, top and right edge of each volume is stored (in this order). This means a row of the edge texture is a sequence like this one: left_edge0, top_edge0, right_edge0, left_edge1, top_edge1, ... See figure 4 which shows a 2×2 mesh which also includes the corresponding ghost cells.

The numbering scheme means that the i-th volume (the ghost volumes must also numbered), which will be accessed using its 2D coordinates (u, v) inside the $sizeX \times sizeY$ texture, with $(u, v) = \left(i - (sizeX \times \lfloor \frac{i}{sizeX} \rfloor, \lfloor \frac{i}{sizeX} \rfloor\right)$ will have to access its edges according to the coordinates which are graphically shown in Figure 3 b).

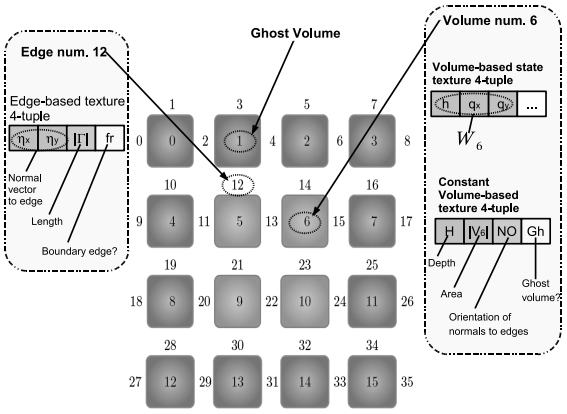

Fig. 4. Arrangement and structure in 2D textures for a 2×2 structured mesh

B. Mapping the computing phases to the GPU

The programable computational steps on a GPU based system correspond to vertices and fragments (potential pixels) processing. The processing units associated to fragments have traditionally been faster but this is not the case on modern GPUs. The way to create the computational units associated to edges and volumes is to either create a vertex or fragment associated to each edge/volume and *draw* them. The computational process is performed by assigning the corresponding input data and the processing code to these elements and drawing them.

We have chosen the traditional approach of drawing a full screen rectangle with a 1 to 1 relation of fragments and edges or volumes, depending on the computation stage. This means one fragment per edge or volume will be drawn and that this will make the code associated to each edge or volume to be run by the GPU. By assigning the correct texture coordinates, each fragment will be able to access the data stored in the before mentioned textures. The texture coordinates of each fragment will be exactly the coordinates of the associated volume or edge in the corresponding textures.

In Figure 5 the computing scheme at each fragment is represented. In this example each fragment performs a computation associated to a volume. Each fragment(volume) accesses both the data related to its four corresponding edges and the data related to the volume itself. All fragments, using the same code, compute a RGB value in parallel which in this case is the new state of each volume.

In Figure 6, the computing phases which are performed on the GPU and the communication points between CPU and GPU are shown. Each GPU computing phase, except the minimum computation, must be performed in a data parallel fashion following a *fragment shader* written in Cg [4]. The same Cg code is applied to each fragment (volume-based or edge-based) and other textures can be accessed to obtain input data. These computing phases are:

1. Edge-based calculations: This phase re-

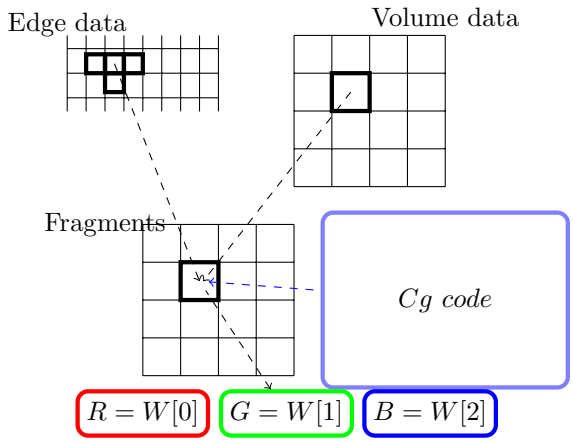

Fig. 5. Computing scheme in a volume-based fragment: for each fragment an instance of the Cg code is run. At each instance other textures can be accessed to obtain input data.

quires the use of the Multiple Rendering Target capabilities of GPUs. This allows to output more than one 4-tuple (color) at one rendering step and increases the arithmetic intensity of the process because otherwise, the computation of the aforementioned results of this step would have to be computed in three steps which would have an impact on efficiency. As a result of this phase, two volume-based textures (one 4-tuple per volume) must be generated: one to store the values M_i for each volume and one to store the values Δt_i .

2. Computation of the local Δt for each volume: For this, a per-volume fragment shader (one fragment per volume is processed) is invoked.

3. Computation of Δt^n: This is a reduction operation which is more complicated to implement on GPUs than on other parallel architectures. However the procedure is based in a recursive decomposition of the minimum and is described in subsection V-C.

4. Computation of W_i^{n+1}. As before, a per-volume fragment shader must also invoked.

5. Update ghost volumes: the state of each of these volumes is obtained from adjacent volumes to impose the corresponding boundary conditions.

The CPU runs a *driver* program which initializes the GPU textures and must control the finalization of the time stepping process while the GPU runs the main calculation phases (see Figure 6).

C. Minimum computation in the GPU

The term *uniform stream reduction* (or simply *stream reduction*, SR in what follows) is used in the related literature to mean an algorithm or processing step which computes a single floating point value from a vector of floating point values. Typical operations which fall in this category are the computation of minimum, maximum or sum of the elements of such a vector, or other associative operations. We have implemented SR functionality in our system, as we need to compute the minimum of a stream.

This results in an iterative algorithm which, at each step, reduces (halves) the size of the array until

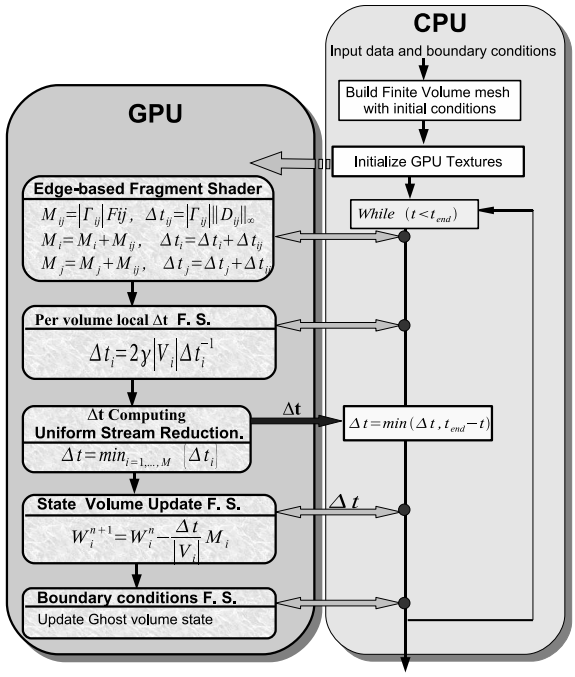

Fig. 6. CPU-GPU program structure

the array degenerates in a single 4-float tuple, which is the resulting value, or from which the resulting value is easily obtained. This scheme takes advantage of the parallel processing capabilities of these devices. If we assume that, at one step array size is $m \times m$ (m rows and m columns), with $m = 2^s > 1$, then this step outputs an array of size $2^{s-1} \times 2^{s-1}$. In such a step, each block of 2×2 input array elements is processed by a fragment processor, resulting in a single element (with the minimum, maximum or sum of input block elements) which is stored in the output array. If the input array size is not a power of two, then this input array is embedded (by filling with an adequate value) in a larger one fulfilling this property. The algorithmic complexity is $O(\log(n))$

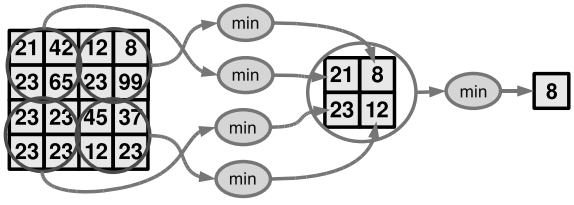

Fig. 7. Stream reduction in GPU

VI. Numerical Experiments

In order to test the solvers, we have considered a problem of an unsteady flow in a $1\ m \times 10\ m$ rectangular channel with a depth function $H(x, y) = 1 - \cos(2\pi x)/2$ and the initial condition is given by $W_i^0(x, y) = \left[h^0(x, y), 0, 0\right]^T$, where:

$$h^0(x, y) = \begin{cases} H(x, y) + 2 & \text{if } x < 5, \\ H(x, y) & \text{other case.} \end{cases}$$

Six uniform meshes of the domain, Q_k, k =

$0, \ldots, 5$, are constructed such that the number of volumes of mesh Q_k is given by $2^{2k} \cdot 10^3, \quad k = 0, \ldots, 5$.

The numerical scheme is run in the time interval $[0, 5]$ except for mesh Q_5 which is solved for the time interval $[0, 0.1]$. The CFL parameter is $\gamma = 0.9$ and wall boundary conditions are considered ($\mathbf{q} \cdot \eta = \mathbf{0}$). Table I shows the execution times for the meshes considered on several platforms. The CPU implementation has been optimized to exploit the SSE CPU units through the use of the Intel Performance Primitives 4.1 (see [2]) and an Intel C++ compiler with em64t extension and the choice -O2 has been used. The CPU executable has been run on an Intel Xeon Nocona 2.66 Ghz. The GPU implementation has been run on three NVIDIA GeForce 8 series cards: 8800 Ultra, 8800 GTX, and 8400M (laptop) associated to CPUs of similar performance through a PCI-express port. The GPU implementation is based on OpenGL together with several fragment shaders written in Cg language.

Figure 8 shows the evolution of the speedup obtained when the GPU is used with respect to the optimized CPU solver when the problem size is increased. The results show that drastic performace benefits can be obtained by efficiently using GPUs as a computing platforms. In fact, a speedup greater than 100 is achieved on both, the NV GF 8800 Ultra and the NV GF 8800 GTX for meshes of practical interest, and even a video card embedded in a laptop, the NV GF 8400GM (about 50$), allows us to obtain a speedup greater than 4. To obtain speedups similar to that obtained with NV GF 8800 cards on a conventional multiprocessor platform, we would need a high number of processors and, as a consequence, a much higher investment would be required.

TABLA I

Execution times for several meshes and GPUs

$Q_k \times t_{end}$	CPU	$8800U$	$8800G$	$8400M$
$Q_0 \times 5.0$	1.05	0.53	0.41	2.25
$Q_1 \times 5.0$	8.09	1.11	0.82	6.4
$Q_2 \times 5.0$	64.23	2.6	1.95	24.7
$Q_3 \times 5.0$	510.6	8.13	6.6	140
$Q_4 \times 5.0$	4046	38.96	41.32	998.6
$Q_5 \times 0.1$	661	5.3	6.4	152.2

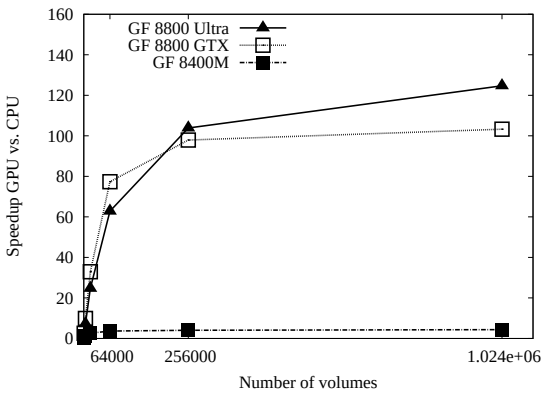

Fig. 8. Speedup obtained with respect the CPU code.

Using the considered meshes, we have also performed several numerical experiments to study the effects of the single precision arithmetic of the GPU on the numerical solution. These experiments do not reveal significant differences between the approximations obtained with a CPU double precision implementation and those obtained with our GPU solver.

VII. Conclusions and Further Work

An efficient first order well balanced finite volume solver of one layer shallow water systems which is able to exploit efficiently the parallel processing power of graphics processing units has been derived. Simulations on an NVIDIA GeForce 8800 GPU (less than 700$) are found to be up to two orders of magnitude faster than the SSE-optimized CPU version of the solver for medium-size problems.

As a further work, we are approaching several lines: a) to extend the strategy to enable efficient simulations on irregular and nonstructured finite volume meshes and for the simulation of two-layer shallow water systems, b) the development of efficient high order solvers for GPUs [3], and c) to use a cluster of CPU-GPUs to enable the fast simulation of realistic large domains with very fine meshes.

Acknowledgements

J. Mantas acknowledges partial support from DGI-MEC project MTM2005-08024. J. Mantas, M. Lastra and C. Ureña also acknowledge partial support from DGI-MEC project TIN2004-07672-c03-02. M. Castro acknowledges partial support from DGI-MEC project MTM2006-08075.

References

[1] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, C. Parés, *A parallel 2d finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows*, Comput. Methods Appl. Mech. Engrg. 195 2788-2815 (2006).

[2] M. J. Castro, J.A. García-Rodríguez, J. M. González, C. Parés, *Solving shallow-water systems in 2D domains using Finite Volume methods and multimedia SSE instructions*, Journal of Comp. and App. Mathematics, 2007.

[3] M.J. Castro, E.D. Fernández-Nieto. A.M Ferreiro, J.A. García-Rodríguez, C. Parés. *High order extensions of Roe schemes for two dimensional nonconservative hyperbolic systems.*, Submitted to J. Sci. Comp. (2008).

[4] Randima Fernando, Mark J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley (2003).

[5] J.M. Gallardo, C. Parés and M. Castro, *On a well-balanced high-order finite volume scheme for shallow water equations with topography and dry areas*, J. Comput. Phys., 227: 574-601, 2007.

[6] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Tim Purcell, *A Survey of General-Purpose Computation on Graphics Hardware*, Eurographics 2005 State of the Art Report (2005).

[7] T.R. Hagen, J.M. Hjelmervik, K.-A. Lie, J.R. Natvig, M. Ofstad Henriksen, *Visual simulation of shallow-water waves*, Sim. Modelling Pract. and Th. 13 (2005) 716-726.

[8] V. Kumar,A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings (2003).

[9] http://www.nvidia.com,

[10] C. Parés, *Numerical methods for nonconservative hyperbolic systems. A theoretical framewok.*, SIAM J. Num. Anal. 44(1): 300-321 (2006).

[11] Rumpf M., Strzodka R., *Graphics Processor Units: New Prospects for Parallel Computing*, L. N. in Computational Science and Engineering, 51, 89-121 (2006).