Simulation of Shallow-Water systems using Graphics Processing Units

Miguel Lastra [a] José M. Mantas [a] Carlos Ureña [a]
Manuel J. Castro [b] José A. García-Rodríguez [c]

[a] *Depto. de Lenguajes y Sistemas Informáticos.*
E.T.S. Ing. Informática y Telecomunicaciones, Univ. Granada, 18071 Granada. Spain.
mlastral@ugr.es, jmmantas@ugr.es, curena@ugr.es

[b] *Depto. de Análisis Matemático*
Facultad de Ciencias, Univ. de Málaga. 29071. Málaga. Spain.
castro@anamat.cie.uma.es

[c] *Dpto. de Matemáticas,*
Universidad de A Coruña, Campus de Elviña s/n, 15071 A Coruña. Spain.
jagrodriguez@udc.es

Abstract

This paper addresses the speedup of the numerical solution of shallow-water systems in 2D domains by using modern Graphics Processing Units (GPUs). A first order well-balanced finite volume numerical scheme for 2D shallow water systems is considered. The potential data parallelism of this method is identified and the scheme is efficiently implemented on GPUs for one-layer shallow-water systems. Numerical experiments performed on several GPUs show the high efficiency of the GPU solver in comparison with a highly optimized implementation of a CPU solver.

Key words: Shallow-water simulation, General Purpose computation on Graphics Processing Units (GPGPU), High performance scientific computing.

1 Introduction

Our goal is to efficiently simulate one-layer fluids that can be modelled 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, such as the simulation of rivers, channels or dambreak problems. However, these simulations impose great demands on 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 are presented in [4]. 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 [5]). Although these improvements have made it possible to obtain results in faster computational times, the simulations still require too much runtime despite the efficient use of all the resources of a powerful PC cluster.

A currently available cost effective emerging architecture is capable of achieving considerable acceleration of 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 speed up the numerical solution of several mathematical models in science and engineering (see [16,20] for a review of the topic). Modern GPUs offer over 100 processing units optimized for massively performing floating point operations in parallel with 4-tuples or 4x4 matrices of floating point numbers (also with smaller tuples and matrices) and floating point operations in general [15]. As a consequence, for several algorithmic structures, these architectures are able to obtain a substantially higher performance than can a powerful general purpose CPU.

In [12], a explicit central-upwind scheme is implemented on a NVIDIA GeForce 7800 GTX card to simulate the one-layer shallow-water system and a speedup from 15 to 30 is achieved with respect to a CPU implementation running on an Intel Xeon processor.

We propose a strategy to design an efficient implementation of the numerical scheme presented in [4] on GPUs using OpenGL [18] and Cg [8]. To do so, it was necessary to adapt the calculations and the data domain of the numerical algorithm to the graphics processing pipeline. A utility library was developed, facilitating the mapping and simplifying the description of the GPU program as a sequential composition of data parallel modules.

The next section describes the structure of the one-layer shallow-water system. Section 3 introduces the underlying numerical scheme. The design and implementation of the GPU version of the numerical solver is described in Section 4. Section 5 presents and analyzes the results obtained when the solver is applied to several meshes using several GPUs. Finally, Section 6 summarizes the main conclusions of the work and presents the lines for further work.

2 Mathematical model: One-layer shallow-water system

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

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

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 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.

System (1) can be formulated as a particular case of a general problem consisting of a system of conservation laws with source terms as follows:

$$\frac{\partial W}{\partial t} + \frac{\partial F1}{\partial x}(W) + \frac{\partial F2}{\partial y}(W) = S_1(W)\frac{\partial H}{\partial x} + S_2(W)\frac{\partial H}{\partial y} \tag{2}$$

where

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

$$S_1(W) = \begin{bmatrix} 0 \\ gh \\ 0 \end{bmatrix}, \; S_2(W) = \begin{bmatrix} 0 \\ 0 \\ gh \end{bmatrix}.$$

Let $J_i(W) = \dfrac{\partial F_i}{\partial W}(W)$, $i = 1, 2$, which denote the Jacobians of the fluxes F_i, $i = 1, 2$. Given a unit vector $\boldsymbol{\eta} = (\eta_x, \eta_y) \in \mathbb{R}^2$, we define the matrix

$$A(W, \boldsymbol{\eta}) = J_1(W)\eta_x + J_2(W)\eta_y,$$

and the vectors

$$F(W, \boldsymbol{\eta}) = F_1(W)\eta_x + F_2(W)\eta_y, \quad S_{\boldsymbol{\eta}}(W) = \eta_x S_1(W) + \eta_y S_2(W).$$

The problem consists in studying the time evolution of $W(x, y, t)$ satisfying System (2).

3 Numerical scheme

In accordance with the description given in [4], this section presents the discretization of System (2) by means of a Finite Volume scheme. First, the computational domain D is divided into L discretization cells or finite volumes, $V_i \subset \mathbb{R}^2$, which are assumed to be closed polygons (here, the volumes are assumed to be quadrangles). Let us denote by $\mathcal{T}$ the set of cells. The following notation is employed: given a finite 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; Γ_{ij} is the common edge of two neighbour cells V_i and V_j, and $|\Gamma_{ij}|$ is its length; $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unit vector which is normal to the edge Γ_{ij} and points towards the cell V_j (see Figure 1).

The approximations to the cell averages of the exact solution produced by the numerical scheme are denoted as follows:

$$W_i^n \cong \frac{1}{|V_i|} \int W(x, y, t^n)\,dxdy \tag{3}$$

where $|V_i|$ is the area of the cell and $t^n = n\Delta t$, with Δt being the time step which for the sake of simplicity is assumed to be constant.

Assume that the approximations at time t^n, W_i^n, have already been calculated. To advance in time, at any time step, a family of projected Riemann Problems in the normal direction to each edge of the mesh is considered. These projected Riemann problems are then linearized by using a path conservative Roe scheme (see [17], [19]). Finally, the approximated solutions of these 1D linear Riemann problems are averaged in the cells to obtain the new piecewise constant approximation of the solution. The resulting numerical scheme is as follows:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{\mid V_i \mid} \sum_{j \in \aleph_i} \mid \Gamma_{ij} \mid F_{ij}^-, \tag{4}$$

where

$$F_{ij}^- = P_{ij}^- (A_{ij}(W_j^n - W_i^n) - S_{ij}(H_j - H_i)), \tag{5}$$

and where $H_l = H(N_l)$ with $l = 1, \ldots, L$, $A_{ij} = A(W_{ij}^n, \boldsymbol{\eta}_{ij})$ and $S_{ij} = S_{\boldsymbol{\eta}_{ij}}(W_{ij}^n)$, with W_{ij}^n an 'intermediate state' between W_i^n and W_j^n. The matrix P_{ij}^- is computed as follows:

$$P_{ij}^- = \frac{1}{2} \mathcal{K}_{ij} \cdot (I - \operatorname{sgn}(\mathcal{D}_{ij})) \cdot \mathcal{K}_{ij}^{-1}, \tag{6}$$

where $\mathcal{D}_{ij}$ is the diagonal matrix whose coefficients are the eigenvalues of A_{ij}, and $\mathcal{K}_{ij}$ is a matrix whose columns are associated eigenvectors. Finally $\operatorname{sgn}(\mathcal{D}_{ij})$ is the diagonal matrix whose coefficients are the sign of the eigenvalues of the matrix A_{ij}.

In the particular case of System (1), A_{ij} and S_{ij} are chosen as follows:

$$A_{ij} = \begin{bmatrix} 0 & \eta_{ij,x} & \eta_{ij,y} \\ (-u_{ij,x}^2 + c_{ij}^2)\eta_{ij,x} - u_{ij,x}u_{ij,y}\eta_{ij,y} & 2u_{ij,x}\eta_{ij,x} + u_{ij,y}\eta_{ij,y} & u_{ij,x}\eta_{ij,y} \\ -u_{ij,x}u_{ij,y}\eta_{ij,x} + (-u_{ij,y}^2 + c_{ij}^2)\eta_{ij,y} & u_{ij,y}\eta_{ij,x} & u_{ij,x}\eta_{ij,x} + 2u_{ij,y}\eta_{ij,y} \end{bmatrix},$$

$$S_{ij} = \begin{bmatrix} 0 \\ gh_{ij}\eta_{ij,x} \\ gh_{ij}\eta_{ij,y} \end{bmatrix},$$

where:

$$c_{ij} = \sqrt{gh_{ij}}; \; u_{ij,\alpha} = \frac{\sqrt{h_i^n}u_{i,\alpha}^n + \sqrt{h_j^n}u_{j,\alpha}^n}{\sqrt{h_i^n} + \sqrt{h_j^n}}, \; \alpha = x, y; \; h_{ij} = \frac{h_i^n + h_j^n}{2}. \tag{7}$$

3.1 Some remarks

(1) Due to the explicit character of the numerical scheme, a CFL condition must be imposed. In practice, the following condition can be used to

compute the n-th time step:

$$\Delta t^n = \min_{i=1,\dots,L} \left\{ \left[\frac{\sum_{j\in\aleph_i} \mid \Gamma_{ij} \mid \parallel D_{ij}^n \parallel_\infty}{2\gamma \mid V_i \mid} \right]^{-1} \right\} \tag{8}$$

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

The time step computed from this criterion can be small during the calculations, which means that a large number of time iterations might be necessary for large time simulations. Although implicit numerical schemes are known to allow for large time steps, here explicit numerical schemes are considered for the following reasons:

(a) We are interested not only in reaching steady states but also in the simulation of fast waves, such as moving shocks or dry/wet fronts appearing in fluvial or coastal hydraulics.

(b) Explicit schemes impose fewer memory overheads, as complex iterative matrix solvers are not required.

(2) As in the case of systems of conservation laws, when sonic rarefaction waves appear it is necessary to modify the Approximate Riemann Solver in order to obtain entropy-satisfying solutions. The Harten-Hyman Entropy Fix technique (see [13]) can easily be adapted to this numerical scheme.

(3) 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 [7]. Extensions to purely non-conservative hyperbolic systems, like the two-layer shallow-water system have also been performed (see [5,19]).

(4) If the numerical scheme (4), is applied without modifications to simulate one-layer flows in which wet/dry fronts appear, the results are not satisfactory: the gradient of the bottom function H generates spurious pressure forces that can make the fluid go up steps or slopes in a nonphysical way. In [1], [2] and [3], some modifications of the numerical scheme have been proposed to avoid this difficulty.

A more sophisticated high order numerical treatment of the wet-dry fronts is introduced in [10]; this ensures the positivity of the water depth at the front as well as the well-balanced properties of the original first-order scheme.

In this paper, we use the modification proposed in [2]: let us suppose that an emerging bottom situation such as the one shown in Figure 2 arises at the edge Γ_{ij}, where $h(N_i) = 0$ and $h(N_j) \leq H(N_j) - H(N_i)$. In this case, instead of using (5), F_{ij}^- is given by:

$$F_{ij}^- = [0, 0, 0]^T .$$

Conversely, in the situation shown in Figure 3, where $h(N_i) \leq H(N_i) -$

$H(N_j)$ and $h(N_j) = 0$, F^-_{ij} is given by:

$$F^-_{ij} = \begin{bmatrix} 0 \\ \frac{1}{2} g (h^n_i)^2 \eta_{ij,x} \\ \frac{1}{2} g (h^n_i)^2 \eta_{ij,y} \end{bmatrix} - F(W_i, \boldsymbol{\eta}_{ij}).$$

In other cases, F^-_{ij} is still given by (5). In [2] it was shown that this modified numerical scheme exactly solves stationary solutions corresponding to water at rest including wet/dry transitions.

4 Obtaining a GPU implementation

To derive an efficient GPU implementation of the numerical scheme, several stages of a methodical approach are completed. This approach may be applied to obtain GPU implementations in other areas of scientific computing and includes the following actions:

- **Data Parallelism identification**:
  From a general point of view, a GPU works by applying the same calculation on a set of input data items to produce a set of output data items. This calculation is usually called a *kernel* [16]. Since a GPU computes bundles of data items in parallel, the calculation for each data item need not depend on the output data obtained for other data items. Therefore a GPU makes it possible to exploit the *data parallelism* [14], that results from concurrently applying identical operations on different data items. Consequently, a data parallel programming model is implemented to efficiently exploit a GPU. In the data parallel programming model, the computation must be organized in computing phases in each of which a high degree of potential data parallelism is exhibited. The data which is managed in each phase may be different but each phase must be described as the application of the same operations to each data item simultaneously.
  Taking this into account, we must now identify the sources of data parallelism in the numerical scheme to be implemented, stablishing which calculations can be made in a data parallel style. Then, each of these calculations must be described as parallel computations and combined sequentially to obtain a data parallel algorithm (see Figure 4).
- **Design of the data storage scheme to maximize the data locality**.
  In a GPU, each processing element has a small cache memory. In order to preserve effective high arithmetical intensity and obtain good performance, the data stored in this memory must be accessed frequently and

data accessing to a slower memory must be minimized. This goal is required in order to obtain a high data locality and involves performing many more accesses to fast memory than to slow memory. To achieve this, it is fundamental to arrange data items in floating point 2D GPU memory arrays in such a way that the computation associated to a data item can be performed by only accessing input data items located in contiguous positions in the texture. In order to choose a data storage which preserves the locality, the data access pattern of each computing phase must be analyzed. Subsection 4.2 describes an efficient storage scheme to perform this on a GPU platform.

- **Description of the data parallel modules for each computing phase**.
  In many cases the whole computational process has to be divided into several steps, which is useful in terms of both design clarity and efficiency. For each computing step a kernel is created and a GPU processing module is associated to each kernel. Each of these modules uses some data as input and produces an output which may be either the final result or an intermediate one to be used by the next module. When GPUs are used, the need to divide the process into several steps is often a requirement imposed by the architecture because the platform limits the number of results that can be produced within a single computing step.
- **Module composition**.
  As the computational process is divided into steps, a CPU driver program is required. This program performs the initialization required by the GPU, binds the data stored on RAM with the parameters of each kernel (this requires the data to be transferred to the GPU memory) and starts the computation of each GPU module.

The use of these stages to derive an efficient GPU implementation of the method described in Section 3 is explained in the following subsections.

4.1 Deriving a data parallel numerical algorithm

A data parallel algorithm was designed on the basis of the mathematical description of the numerical scheme. Figure 4 shows a graphical description of the parallel numerical algorithm. In this figure, the main calculation phases are identified with circled numbers and the main sources of data parallelism are clearly indicated. A `Parfor each` <*data_item*> block denotes that the calculation affected by it can be performed simultaneously for each data item of a set (in this algorithm, the data items can represent the volumes or the edges of the mesh). The arrows connecting two computing phases represent data dependences between the two phases.

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 (4) must be evaluated to update the state of each cell. Of course, the data computed at the $(n+1)$-th time step does present a degree of dependence on the previous one, but this does not restrict the parallelization of the computations performed at a particular time step. In fact, the four main calculation phases of the evaluation present a high degree of parallelism and must be completed consecutively, as follows:

(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 $M_{ij} =\mid \Gamma_{ij} \mid F_{ij}^{-} \in \mathbb{R}^3$ must be computed as the contribution of each edge to the sum associated to the corresponding neighbour cells V_i and V_j (see Equation (4)). This contribution can be computed independently for each edge and must be added to the partial sums associated to each cell (M_i and M_j). This 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 only the data corresponding to the volumes V_i and V_j are needed to compute the contribution for one particular edge Γ_{ij}, this computation presents a high arithmetic intensity and locality.
   b) The value $\Delta t_{ij} =\mid \Gamma_{ij} \mid \| D_{ij}^{n} \|_\infty$ must be computed and added to the partial sums associated to each cell (Δt_i and Δt_j) as an intermediate step to compute the n-th time step Δt^n (see Equation (8)).

   Both calculations for an edge can be computed simultaneously with respect to the calculations associated to other 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 according to Equation (8). 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 [14].
(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 in the previous phases. This phase can also be completed in parallel (see Figure 4).

Let us note the following, 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 level of arithmetic intensity and the computation

exhibits a high degree of locality, because the computation for each edge or volume only depends on the data from adjacent volumes;

c) the scheme exhibits a high degree of potential data parallelism (see Figure 4) because the computation at each edge or volume is independent with respect to that performed or associated to the other edges or volumes.

Thus, this problem seems suitable for implementation on modern GPUs. 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 to perform some mappings between the problem domain and this pipeline.

4.2 Data storage and arrangement in the GPU

In computer graphics most of the data is represented by 3 or 4-tuples (`float3` and `float4` data types in the Cg language [8]) to denote points and vectors. Transformations are usually represented by 3×3 or 4×4 matrices. These matrices are multiplied to obtain the composition of different transformations and vectors and the points are multiplied by these matrices to apply the transformations. Therefore these types of data storage and operations are highly optimized on graphics hardware.

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 is mainly used (in graphics applications) to store colours representing an image to be applied to a 3D object.

The aforementioned mechanisms enable 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.

4.2.1 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 for the i-th volume V_i ($i = 1, \ldots, L$) is the following:

(1) The evaluation of the depth function for the volume (H_i).
(2) The area of the volume ($|V_i|$).
(3) The information about the orientation of the normal vector associated to each edge of the volume.

(4) An indication as to whether the volume is a ghost volume (ghost volumes are fictitious cells which are only used to impose the boundary conditions, see subsection below for a more detailed description).

This data can be packed in a vector of 4 floating point numbers (`float4` data type): one floating point value for H_i, one for $|V_i|$, one for the normal orientation information and one to mark ghost volumes (see Figure 6). The penultimate term requires some additional explanation 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 orientations of the normal vectors, is treated as an integer value and the orientation of the vectors is obtained by evaluating the value of its four least significant bits. A value of 1 corresponds to the normal vector pointing towards the exterior of the volume. The mapping between the bits and edges is shown in Figure 5a.

These 4-tuples are stored in a $n\times m$ texture where $n\times m$ is at least equal to the number of volumes, including several ghost volumes (see Figure 6). This texture is a rectangular matrix where each position contains a 4-tuple and is associated to a volume (wether a ghost or not). The volumes are arranged in the texture following the column-major ordering. This numbering scheme means that the i-th volume (bearing in mind that the ghost volumes must also be numbered), is accessed using its 2D coordinates (u, v) inside the $n \times m$ texture, with:

$$(u,v) = \left(i - n \times \left\lfloor \frac{i}{n} \right\rfloor, \quad \left\lfloor \frac{i}{n} \right\rfloor\right). \tag{9}$$

Figure 6 shows this ordering for a 2×2 mesh where the final mesh has 4×4 cells because the corresponding ghost cells have been included.

- The volume state data include 3 floating point values. Each W_i^n vector represents the state of the i-th volume at the n-th time step. These data are stored in another texture with an identical numbering scheme as the constant volume-based texture (see Figure 6).

With this arrangement for the volume information, the data associated to neighbour finite volumes is stored in contiguous positions in the 2D volume-based textures. As a consequence, the data locality is enhanced.

4.2.2 Ghost cells

As it has been mentioned above, constant data for each cell or volume includes an indication (a logic value) with information about whether the cell is a *ghost cell* or not. These ghost cells are used to impose the boundary conditions. We have tested our current implementation on rectangular domains, and we have used ghost cells for boundary rows and columns (the first and last row and column).

The GPU is responsible for allocating available hardware processors to kernel instances, in such a way that the hardware is efficiently used, because as soon as a processor ends running a kernel instance, it is assigned to other data items, thus no processor is idle while the whole computation step has not been finished (that is, while not all cells have been processed). During a computation step, this processor to cell assignment is done once for every cell, including ghost cells. In order to take into account these ghost cells, the kernel must be extended with a check at its beginning. In this check, the above mentioned ghost cell indicator is read. If it is active (that is, if the cell is a ghost cell), then a special cancel instruction is executed, and a constant value (or no output value) is written to the output matrix. This instruction tells the control hardware that the kernel instance has ended the computation for this cell.

By using that scheme, almost no computation time is devoted to ghost cells, because a processor assigned to a ghost cell is quickly switched to another one by the control hardware in the GPU. As a result almost all computing time in each step will be employed in non-ghost cells which belong to the original domain. Assuming that the total number of cells is big as compared to the available number of processors, as is usually the case, the fraction of time employed in ghost cells will be very small.

The can also be used to model a domain with non-rectangular boundaries. In this case, the domain is also composed of rectangular cells in a grid, however additional padding ghost cells are added around the initial domain cells, in order to embed the original non rectangular domain in a rectangular matrix, which can be easily mapped into GPU textures as usual.

4.2.3 Edge-based information texture

The information about an edge, Γ, that must be stored is: the normal vector 2D coordinates, (η_x, η_y), its length, $|\Gamma|$, and a value that indicates whether or not the edge is a boundary edge or not. Again, this information can be represented by a rectangular texture of 4-tuples. The fourth number is set to be positive on edges which are at the boundary of the volume mesh.

The edge information texture has the same number of rows as a volume-based texture, but more columns because the number of edges is greater. At each row, for each volume, from left to right, the information on the left, top and right edge of each volume is stored (in this order). This means a row of the edge texture takes the form of the following sequence (see figure 6):

$$left\_edge0, \quad top\_edge0, \quad right\_edge0, \quad left\_edge1, \quad top\_edge1, \quad \ldots$$

To perform several computations associated to the i-th volume whose state is stored at position (u, v) of the volume state texture, we must access information about its edges. This information is stored in the edge-based texture with the coordinates which are graphically shown in Figure 5b and which corresponds to the neighbour cells in the edge-based texture (see Figure 7).

With this numbering scheme (for arranging data about volumes and edges), the most costly calculations of the numerical scheme can be performed on the GPU exhibiting a high degree of data locality because all the data necessary to compute a value associated to a volume or an edge is located in contiguous 2D coordinates.

4.3 Mapping the computing phases to the GPU

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

In this article, the traditional approach of drawing a full screen rectangle with a 1 to 1 relation of fragments and edges or volumes (depending on the computing phase) has been adopted. Thus, one fragment per edge or volume is drawn, and so the code associated to each edge or volume can be run by the GPU. By assigning the correct texture coordinates, each fragment will be able to access the data stored in the above mentioned textures. The texture coordinates of each fragment are exactly those of the associated volume or edge in the corresponding textures.

Figure 7 shows the computing scheme at each fragment. In this example each fragment performs a computation associated to a volume, accessing 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 (three floating point numbers representing the Red, Green and Blue components of a colour) value in parallel which is the new state of each volume. In this case, the RGB floating point numbers do not represent a colour but, rather, the values involved in the computation explained here.

Figure 8 shows the computing phases performed on the GPU and the communication points between the CPU and the GPU. Each GPU computing phase, except the minimum computation, must be performed in a data parallel fashion following a *fragment shader* written in Cg [8]. The same Cg code is applied to each fragment (volume or edge-based) and other textures can be accessed to obtain input data. These computing phases are as follows:

1. Edge-based calculations: this phase requires the use of the Multiple Rendering Target capabilities of the GPUs [18,20]. This makes it possible to output more than one 4-tuple (colour) at one rendering step and increases the arithmetical intensity of the process; otherwise, these results would have to be computed in three steps which would have a negative 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 M_i values for each volume and one to store the Δt_i values.

2. Computation of the local Δt for each volume: to do this, a per-volume fragment shader (one fragment per volume is processed) is invoked to compute the texture which stores the Δt_i values ($i = 1, \ldots, L$).

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 on a recursive decomposition of the minimum, as described in subsection 4.3.1.

4. Computation of W_i^{n+1}: as before, a per-volume fragment shader, performing the calculation described in phase (4) of Figure 4, must also be invoked to compute the new volume state texture.

5. Update ghost volumes: the state of each ghost volume is obtained from adjacent volumes to impose the corresponding boundary conditions. For instance, to impose wall boundary conditions ($\mathbf{q} \cdot \eta = \mathbf{0}$), for each ghost cell V_g, its state is set to $(h_k, 0, 0)^T$, where h_k is obtained from the state of the non ghost cell V_k which neighbours V_g.

4.3.1 Minimum computation in the GPU

The term *uniform stream reduction* (or simply *stream reduction*, SR below) is used to describe an algorithm or processing step which computes a single floating point value from a large vector of floating point values. Typical operations which fall in this category are the computation of the minimum, maximum or sum of the elements of such a vector, or other associative operations. SR functionality has been implemented in the system described here, as the minimum of a texture must be computed.

The fact that these operations are associative enables efficient parallel schemes to be implemented, as different parts of the input data set can be processed independently. In the context of GPUs, it has been shown that it is efficient to make each fragment shader work on a 2×2 block of adjacent data elements (each element being a 4-float tuple) and to compute a single resulting tuple from those (with the minimum, maximum, sum, etc. of the input tuples). This operation is included in a multistage algorithm. At each stage, a texture with $2^k \times 2^k$ tuples is processed by $2^{k-1} \times 2^{k-1}$ independently running instances of a fragment shader, yielding an output texture with $2^{k-1} \times 2^{k-1}$ tuples, which is used in turn as the input for the next step. The process ends with a 1×1 texture, containing a single 4-float tuple from which the final result is obtained. If the input texture size is $n = 2^m \times 2^m$, obviously the algorithm performs $m = \log_4(n)$ steps, each one comprising at most n basic operations. Thus, the algorithmic complexity is $O(n \log(n))$, and at each step, all available fragment shaders processors can be used in parallel, as desired.

4.4 Building the final CPU-GPU program

The CPU runs a *driver* program which initializes the GPU textures and controls the finalization of the time stepping process while the GPU runs the main calculation phases (see Figure 8). The driver program is based on a software layer that hides many details about the graphics API (Application Programming Interface) and allows the programmer to concentrate on describing the computational phases of the numerical algorithm.

4.4.1 A software layer to improve programmability

The C++ implementation of the shallow-water simulation program makes use of a set of classes which allow some degree of abstraction concerning the underlying Cg code. As the Cg structure is graphics oriented, it is sometimes cumbersome to map general purpose computation schemes onto this graphics-oriented API. Thus, the need has been identified for an abstraction layer making it possible to easily express general-purpose computing concepts in a C++ program, and hiding, as far as possible, the Cg details from the programmer, thus leading to a shorter, more readable, modifiable and robust code. It is necessary to take into account the joint GPU-CPU working scheme, whereby every GPU kernel must be initiated by an appropriate CPU command, and though which all data stored in the GPU (except the intermediate results) must be transferred between GPU and the CPU using explicit CPU commands.

The design pattern thus implemented comprises two main categories of entities: on the one hand, the data (two-dimensional arrays) and, on the other, the computations acting on that data (for instance, reductions). Each type of entity leads to a C++ class.

The main data structure used is a two dimensional array of 4-float tuples (henceforth, called simply arrays), which can be identified with a stream. Various data related classes have been designed: thus, there is one class for RAM arrays (named `RAMArray2D`), which is a class whose instances encapsulate an array stored in RAM, and another one for arrays stored in the GPUs (named `GPUArray2D`). GPU array class instances hold a Cg handler for an array (a texture) stored in the GPU memory. Straightforward methods are provided for creating, destroying and accessing the 4-float tuples for both types of arrays. The programmer is also provided with the means to transfer arrays between the CPU and the GPU. Additionally, we have found it useful to design a class for a set of equally-sized arrays stored in GPU memory (`MRTGPUArray2D`), as this abstraction allows us to make use of the Cg capabilities for using kernels writing results on more than one output array (a capability often described as *Multiple Rendering Targets*).

Regarding computation, the key idea is that, for each type of operation acting on GPU arrays, an associated C++ *procedural* class should be designed. This class should include methods for transparently creating and destroying any intermediate GPU array, as needed, and should also include methods for starting a computation by running the appropriate kernels (which obviously requires sending parameters to the GPU and naming both the input and the output GPU arrays). All the Cg details involved are hidden from the programmer using the library: any intermediate array needed is created within the constructor. Running the entire computation merely requires a call to the appropriate method for a correctly built instance of the procedural class. These C++ method parameters include any such that may be needed for the computation (this typically includes scalar values or small vectors, and pointers to input GPU arrays). As a result of this call, a new `GPUArray2D` instance (or a set of them, if the `MRTGPUArray2D` is used) is returned to the programmer.

Following this design pattern, we have implemented procedural classes for carrying out uniform stream reduction (class `GPUReductor`). Any other computation comprising kernels running on GPU arrays can be fitted into this scheme. The pattern allows the programmer to easily reuse any intermediate stream space in GPU memory for multiple computations, saving memory and stream creation time. This is easily done because, as stated above, the intermediate arrays are encapsulated within the procedural class instances.

The software component thus created allows for any required initialization of Cg, OpenGL [18], GLUT [18] and related libraries. It also includes auxiliary functions for loading, compiling and running Cg fragment shaders, and for setting their parameters. As a result, using this pattern leads to more easily modifiable, reusable and robust GPU computation programs. Moreover, we believe this pattern also enables an easier transition to other more modern software architectures for GPU programming, such as CUDA [9], because Cg related issues are properly isolated from concepts related to the organization of the computation.

Code extract 6.1 shows a part of the initialization phase. First two fragment shaders are loaded and compiled by giving the file where each fragment shader is stored as a parameter. In the next step four `GPUArray2D` objects are created. They are created from a regular CPU array (called *nodes* in this example), with two numerical values which represent the horizontal and vertical size and a string used as a name. At the end of the code, a `MRTGPUArray2D` (`M`ultiple `R`endering `T`arget GPUArray2D) object is created. This object provides the same functionality as a `GPUArray2D` but can have up to 4 textures attached. This makes it possible to run fragment shaders that produce up to 4 different 4-tuples per fragment.

Code extract 6.2 shows a part of the driver program, using the above mentioned abstraction layer. In the first part, the computation associated to edges is run. This basically involves binding or activating the precompiled fragment shader, setting the parameters to be passed to the fragment shader and letting the computation start by using the `DrawQuad` method associated to the output texture(s).

The computation associated to the edges produces three results per fragment and is therefore run using the above mentioned `MRTGPUArray2D` object. The computation related to the local Δt value is performed using a regular `GPUArray2D`.

It can also be seen that the following two methods for setting the parameter values of the fragment shaders are used. One is used for the texture parameters and the other is for the floating point parameters:

- `BindAndEnableTextureParam`
- `BindAndEnableFloatParam`

Finally code extract 6.3 shows the header of the fragment shader associated to the edge-based computation. The names of the parameters used in code 6.2 have a one to one relationship with the parameters of this header.

5 Numerical Experiments

We have considered two different test problems to show the efficiency and accuracy of the GPU solver. This GPU implementation has been run on three NVIDIA GeForce cards: GTX 280, 8800 Ultra and 8400M GS (laptop) associated to CPUs of similar performance through a PCI-express port. This implementation is based on the previously described software layer (see 4.4.1) together with several fragment shaders written in Cg language.

In order to perform comparisons with a CPU platform, we have used a CPU implementation of the numerical scheme which has been run on an Intel Xeon Nocona 2.66 Ghz. This CPU solver has been optimized to exploit the SSE CPU units through the use of the Intel Performance Primitives 4.1 (see [5]) and has been compiled with an Intel C++ compiler with em64t extension using the choice -O2.

5.1 Test 1

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 $W_i^0(x, y) = [h^0(x, y), 0, 0]^T$, where:

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

Six uniform meshes of the domain, $Q_k, \quad k = 0, \ldots, 5$, 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 1 shows the execution times for the meshes considered on several platforms. Figure 10 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 GF GTX 280 and the GF 8800 Ultra for meshes of practical interest, and even a video card embedded in a laptop, the NV GF 8400GM, allows us to obtain a speedup greater than 4.

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.

5.2 Test 2. Wet/dry front simulation

In this subsection we present a test proposed by Galloüet in [11], based on a numerical test initially proposed by E. Toro in [21]. The purpose of this test is to study the robustness and the efficiency of the numerical scheme considered here when wet/dry fronts appear over a non-flat bottom topography. Let us consider a 25 $m \times 5$ m rectangular channel channel with bottom topography given by

$$H(x,y) = \begin{cases} 9 & \text{if } x > \frac{25}{3} \text{ and } x < \frac{25}{2}, \\ 10 & \text{otherwise.} \end{cases}$$

with initial conditions

$$h^0(x,y) = H(x,y), \quad q_y(x,y) = 0, \quad q_x(x,y) = \begin{cases} -350 & \text{if } x < \frac{50}{3}, \\ 350 & \text{otherwise.} \end{cases}$$

The test consists on two supercritical waves with instantly separate the water column at line $x = \frac{50}{3}$ and produce a vacuum. The wave travelling to the left presents a shock in the water surface, produced by the presence of the bump (see figure 12). Next the left travelling wave interacts with the step, producing also a vacuum over it (see figure 12).

Five uniform meshes of the domain, $Q_k, \quad k = 0, \ldots, 4$, are constructed such that the number of volumes of mesh Q_k is given by $2^{2k} \cdot 3125, \quad k = 0, \ldots, 4$.

The numerical scheme is run in the time interval $[0, 0.65]$. The CFL parameter is set to $\gamma = 0.8$ and wall boundary conditions are considered ($\mathbf{q} \cdot \eta = \mathbf{0}$) at the lateral wall, and free boundary conditions at the open boundaries.

Table 2 shows the execution times for the meshes considered on several platforms and Figure 11 shows the evolution of the speedup obtained with respect to the optimized CPU solver. As can be seen, although the speedup values with respect to the CPU solver for medium-size problems are also considerable, they are less high than the values obtained in Test 1. This is because this test contains states with totally dry cells. While in the CPU solver, these cells require less computing cost than the rest of cells, in the GPU solver all the cells require the same computing cost (due to the restrictions of the GPU computing model).

Figures 12 and 13 show the computed free surface over the mesh Q_2 at a longitudinal cut of the channel at $y = 2.5$ for both, GPU (line with circles) and CPU (line with '+') implementations. A 1D reference solution is computed using a 1D high order solver described in [10] over a uniform mesh composed with 2000 cells. As in the previous test, no significant differences between the GPU and CPU implementation are observed: only small differences close to the wet/dry front can be observed. The main differences with respect to the reference solution are due to the use of a high order numerical scheme, while the numerical scheme considered here is only first order.

It is important to note that to obtain speedups similar to that obtained with the NVIDIA GF GTX 280 and the NVIDIA GF 8800 Ultra 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.

6 Conclusions and Further Work

An efficient first-order well-balanced finite volume solver for one layer shallow-water systems, capable of efficiently exploiting the parallel processing power of graphics processing units, has been derived. The solver has been developed by following several stages of a methodical approach and its implementation is based on an abstraction software layer which hides many problematic details of the Graphics API. Simulations carried out on an NVIDIA GeForce GTX 280 GPU were found to be up to two orders of magnitude faster than an SSE-optimized CPU version of the solver for medium-size uniform problems. These simulations also show that the numerical solutions obtained with the proposed solver are accurate enough for practical applications.

As further work, the following lines are being considered:

- Extending the strategy to enable efficient simulations on irregular and non-structured finite volume meshes and to address the simulation of two-layer shallow-water systems.
- Developing efficient high order solvers for GPUs [6].
- Using a cluster of CPU-GPUs to enable the fast simulation of realistic large domains with very fine meshes.

Regarding the first point above, and as it has been detailed in the previous sections, the simulation method assumes a rectangular cell grid as computational base and this schema can be mapped very efficiently on modern GPUs. There might be of course simulation environments where the water stream bed might not have a rectangular shape. These cases would not require any important change in our implementation as any cell can be marked as boundary (or ghost), letting the stream boundary have an arbitrary shape. In this case, the boundary cell would be recognized by a numerical value (there are unused values in the cell state data) instead of by its grid indices. Finally, as the data used by the GPU must have a rectangular shape, some padding cells would probably have to be added. These cells could be marked as not required and the computation associated to them would be only a discard operation witch makes the GPU skip to the next fragment.

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] P. Brufau, M.E. Vázquez and P. García, A numerical model for the flooding and drying of irregular domains. Int. J. Numer. Meth. Fluids, 39, 247–275, 2002.

[2] Castro M.J, Ferreiro A., García J.A, González J.M, Macías J, Parés C, Vázquez-Cendón M.E, On the numerical treatment of wet/dry fronts in shallow flows: application to one-layer and two-layers systems. Math. Comp. Mod. **42**, 419–439, 2005.

[3] Castro M.J, González J.M, Parés C, Numerical treatment of wet/dry fronts in shallow flows with a modified Roe scheme. Math. Mod. Meth. App. Sci. **16**, 897–931, 2006.

[4] 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. Methods Appl. Mech. Engrg. **195**: 2788–2815, 2006.

[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. App. Math., 221: 16-32, 2008.

[6] Castro, M.J., Chacón, T., Fernández, E.D., González, J. M., Parés, C. Well-balanced finite volume schemes for 2D non-homogeneous hyperbolic systems. Application to the dam-break of Aznalcóllar. Comput. Methods Appl. Mech. Engrg. 197: 3932-3950, 2008.

[7] Castro, M.J., Fernández-Nieto E.D., Ferreiro A.M, García-Rodríguez J.A., Parés C., High order extensions of Roe schemes for two dimensional nonconservative hyperbolic systems, J. Sci. Comput., 39: 67-114, 2009.

[8] Fernando R., Kilgard M.J., The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley, 2003.

[9] nVIDIA, nVIDIA CUDA Component Unified Device Architecture. Programming Guide Version 1.1. Technical report nVIDIA Corporation, 2007.

[10] Gallardo J.M., Parés C., Castro M.J., 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.

[11] Galloüet, T., Hérard, J.M., Seguin, N. Some approximate Godunov schemes to compute shallow-water equations with topography, Comput. Fluids 32: 479513, 2003.

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

[13] Harten A, Hyman J.M, Self-adjusting grid methods for one-dimensional hyperbolic conservation laws. J. Comp. Phys. 50:235–269, 1983.

[14] Kumar V.,Grama A., Gupta A., Karypis G., Introduction to Parallel Computing, Benjamin/Cummings, 2003.

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

[16] Owens J.D, Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A.E., Purcell T., A Survey of General-Purpose Computation on Graphics Hardware, Eurographics 2005 State of the Art Report, 2005.

[17] Parés C, Castro M.J, On the well-balance property of Roe's method for non conservative hiperbolic systems. Applicatons to shallow-water systems..ESAIM: M2AN, 38(5): 821–852, 2004.

[18] OpenGL Architecture Review Board, Shreiner D., Woo M., Neider J., Davis T., OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1, Addison-Wesley Professional, 2007.

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

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

[21] Toro,E.F., Shock-Capturing Methods for Free-Surface Shallow Flows, Wiley, 2001.

$Q_k \times t_{end}$	CPU	$GTX280$	$8800U$	$8400MGS$
$Q_0 \times 5.0$	1.05	0.347	0.53	2.25
$Q_1 \times 5.0$	8.09	0.748	1.11	6.4
$Q_2 \times 5.0$	64.23	1.78	2.6	24.7
$Q_3 \times 5.0$	510.6	5.25	8.13	140
$Q_4 \times 5.0$	4046	23.85	38.96	998.6
$Q_5 \times 0.1$	661	3.07	5.3	152.2

Table 1
Test 1. Execution times for several meshes and GPUs.

Q_k	CPU	$GTX280$	$8800U$	$8400MGS$
Q_0	0.61	0.11	0.11	0.72
Q_1	4.66	0.27	0.29	3.1
Q_2	38.12	0.72	0.9	15.12
Q_3	303.55	2.87	4.24	97.87
Q_4	2512.53	17.1	27.83	737.97

Table 2
Test 2. Execution times for several meshes and GPUs.

```
// Compile prepare the Cg kernels for later use

edgeFS   = LoadAndBindProgram("EdgeFS.cg","main");
nodeFS   = LoadAndBindProgram("NodeFS.cg","main");

// Create and initialize the arrays with data associated
//  to nodes
// We need 2 arrays to store the volume state data as we need
//  information about the current and next state

nodeState[0]   = new GPUArray2D(nodes,sizex,sizey,
                                "NodeState0");
nodeState[1]   = new GPUArray2D(nodes,sizex,sizey,
                                "NodeState1");
nodesConstants= new GPUArray2D(nodesConstants,sizex,sizey,
                                "NodeConstantData");
deltaT         = new GPUArray2D(nodes,sizex,sizey,
                                "LocalDeltaT");

// Create and initialize the arrays with data
//  associated to edges

edgeConstants=new GPUArray2D(edges,edge_sizex,edge_sizey,
                             "EdgeData");

// Create a MRTGPUArray that is able to hold up to 4 textures
//  allowing to hold the results generated by MRT kernels
// In this example all these arrays are initilized
//  with the same data as it will get overwritten anyway

mrt=new MRTGPUArray2D(3,edge_sizex,edge_sizey,"MRT");
mrt->AddArray(edges);
mrt->AddArray(edges);
mrt->AddArray(edges);
```

Code 6.1. Extract of the driver program initialization phase

```
// Start computation at the edges

BindFragmentShader(edgeFS);

// Bind parameters of the kernel

nodeState[state]->BindAndEnableTextureParam(edgeFS,"nodes");
edgeConstants    ->BindAndEnableTextureParam(edgeFS,"edges");
nodesConstants   ->BindAndEnableTextureParam(edgeFS,"nConstants");
BindAndEnableFloatParam(edgeFS,edge_sizex,"sizex");
BindAndEnableFloatParam(edgeFS,edge_sizey,"sizey");

// Run active Cg program for each fragment.
// One fragment per position of the output array (mrt)
//   will be created

mrt->DrawQuad();


// Start computation local deltaT

BindFragmentShader(nodeFS);

// Bind parameters of the kernel
nodeState[state]->BindAndEnableTextureParam(nodeFS,"nodes");
nodesConstants   ->BindAndEnableTextureParam(nodeFS,"nConstants");

// Bind the 3rd texture of the MRT array

mrt->BindAndEnableTextureParam(nodeFS,"precalc",2);
BindAndEnableFloatParam(nodeFS,gamma,"gamma");
BindAndEnableFloatParam(nodeFS,sizex,"sizex");
BindAndEnableFloatParam(nodeFS,sizey,"sizey");

// Run active Cg program for each fragment.
// One fragment per position of the output array (deltaT)
//   will be created

deltaT->DrawQuad();
```

Code 6.2. Extract of the driver program computing phase

```
void main
(uniform  samplerRECT nodes,      //Node state information
 uniform  samplerRECT edges,      //Edge information
 uniform  samplerRECT nConstans, //Node constant information
 uniform float sizex,             //Horizontal size of the texture
 uniform float sizey,             //Vertical size of texture
 float2 coords : TEXCOORD0,       //Edge coordinates
 out float3 fplus:   COLOR0,      //F+
 out float3 fminus:  COLOR1,      //F-
 out float3 precalc: COLOR2)      //This value will be used to
                                  // compute DeltaT at the nodes
{
 //Fragment shader code
}
```

Code 6.3. Extract of a fragment shader

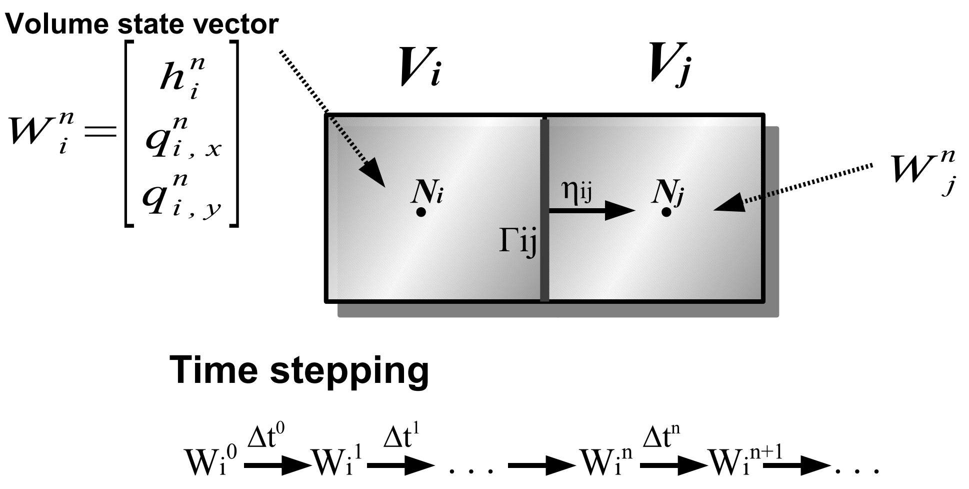

Volume state vector
$W_i^n = \begin{bmatrix} h_i^n \\ q_{i,x}^n \\ q_{i,y}^n \end{bmatrix}$
V_i
V_j
N_i
η_{ij}
N_j
Γ_{ij}
W_j^n
Time stepping
$W_i^0 \xrightarrow{\Delta t^0} W_i^1 \xrightarrow{\Delta t^1} \dots \longrightarrow W_i^n \xrightarrow{\Delta t^n} W_i^{n+1} \longrightarrow \dots$

Fig. 1. Finite volumes

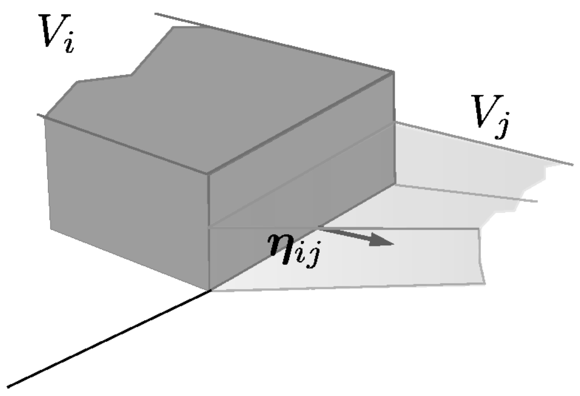


Fig. 2. Emerging bottom situations: case 1

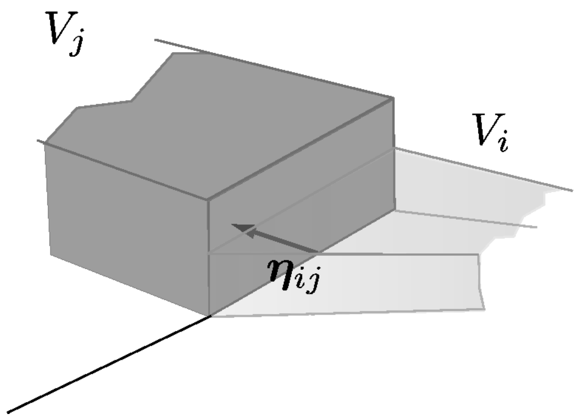


Fig. 3. Emerging bottom situations: case 2

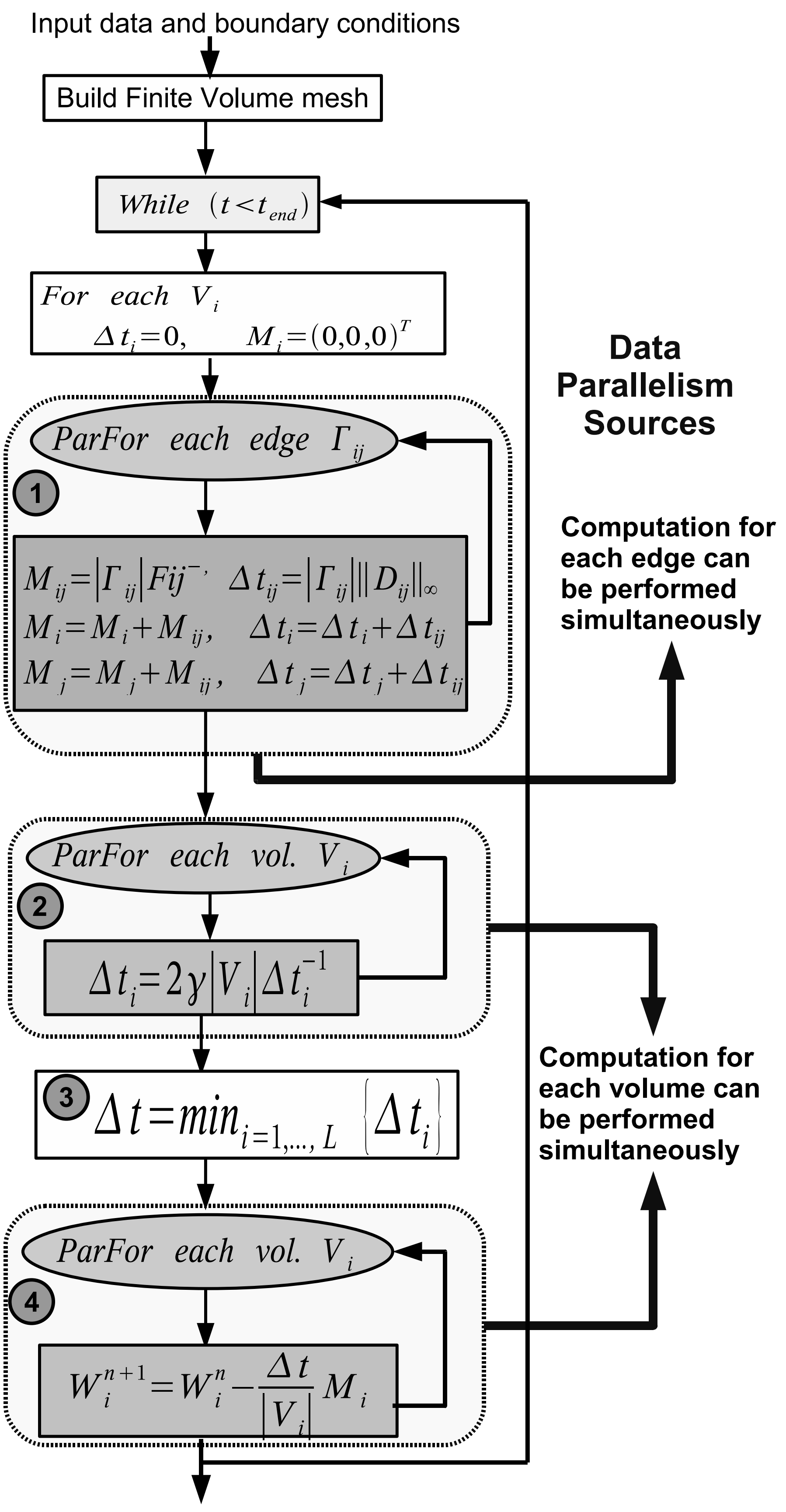

Input data and boundary conditions
Build Finite Volume mesh
$While \ (t<t_{end})$
$For \ each \ V_i$
$\Delta t_i=0, \quad M_i=(0,0,0)^T$
$ParFor \ each \ edge \ \Gamma_{ij}$
1
$M_{ij}=|\Gamma_{ij}|Fij^{-}, \ \Delta t_{ij}=|\Gamma_{ij}|\|D_{ij}\|_\infty$
$M_i=M_i+M_{ij}, \quad \Delta t_i=\Delta t_i+\Delta t_{ij}$
$M_j=M_j+M_{ij}, \quad \Delta t_j=\Delta t_j+\Delta t_{ij}$
$ParFor \ each \ vol. \ V_i$
2
$\Delta t_i=2\gamma|V_i|\Delta t_i^{-1}$
3
$\Delta t=min_{i=1,\ldots,L}\{\Delta t_i\}$
$ParFor \ each \ vol. \ V_i$
4
$W_i^{n+1}=W_i^n-\frac{\Delta t}{|V_i|}M_i$
Data Parallelism Sources
Computation for each edge can be performed simultaneously
Computation for each volume can be performed simultaneously

Fig. 4. Main calculation phases in the parallel algorithm

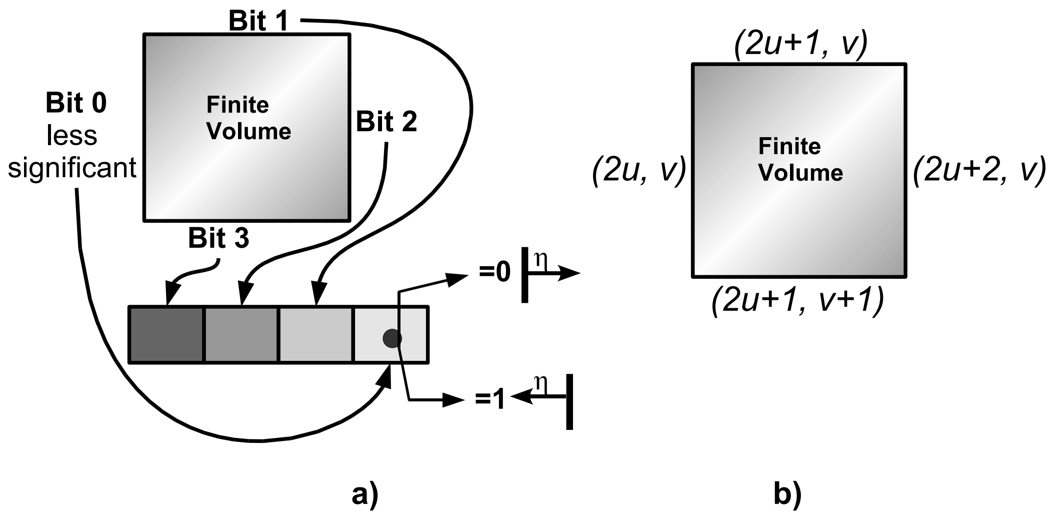


Fig. 5. Edges-bits mapping and edge coordinates for a cell

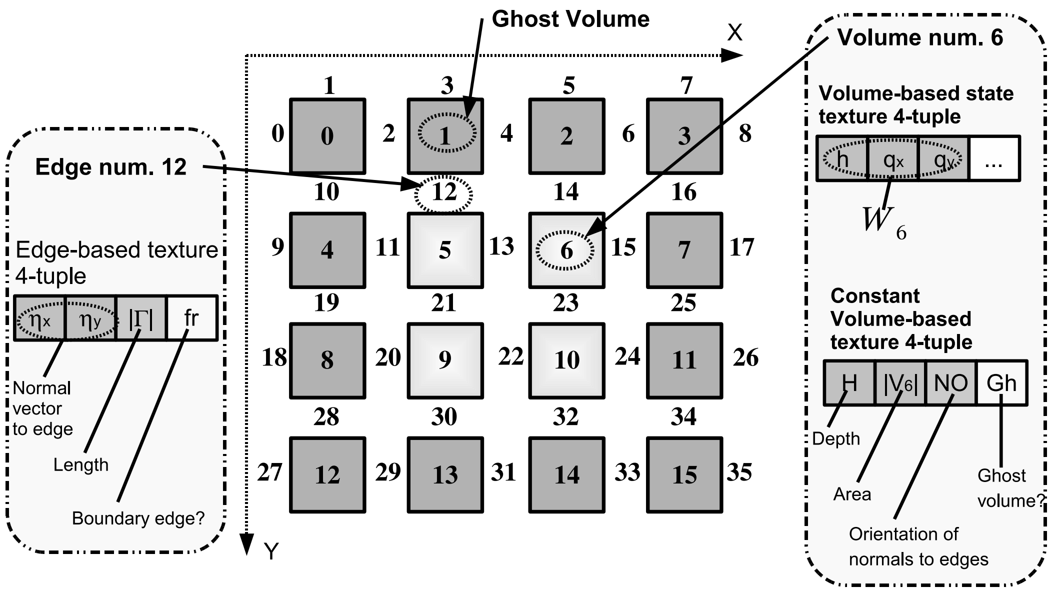


Fig. 6. Arrangement and structure in 2D textures for a 2×2 structured mesh

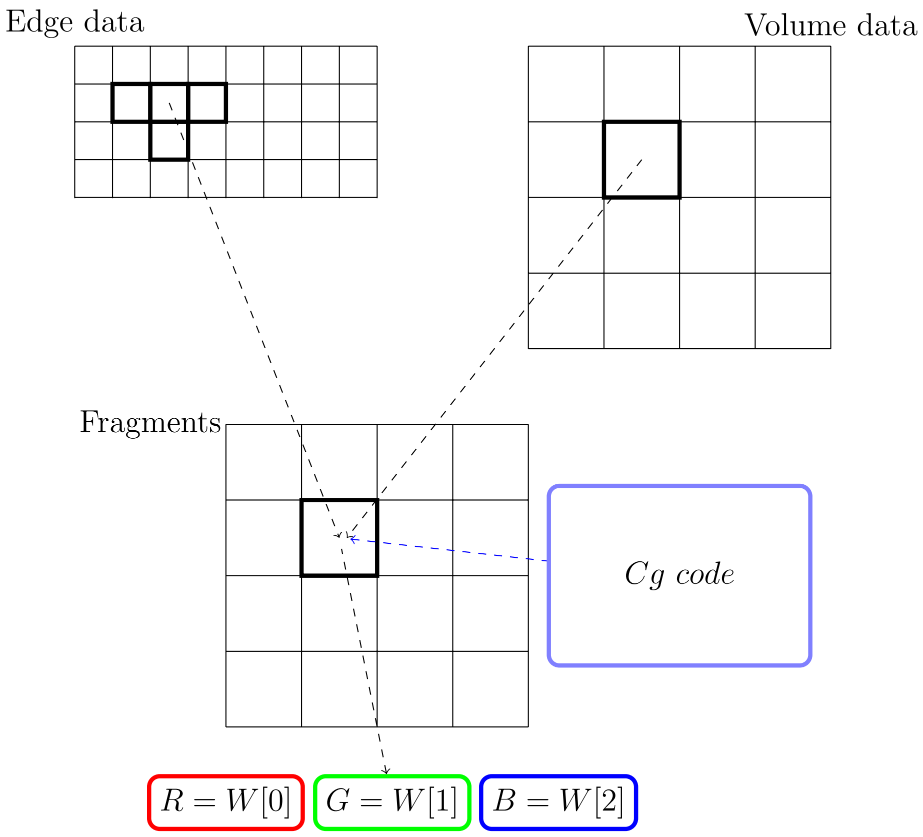
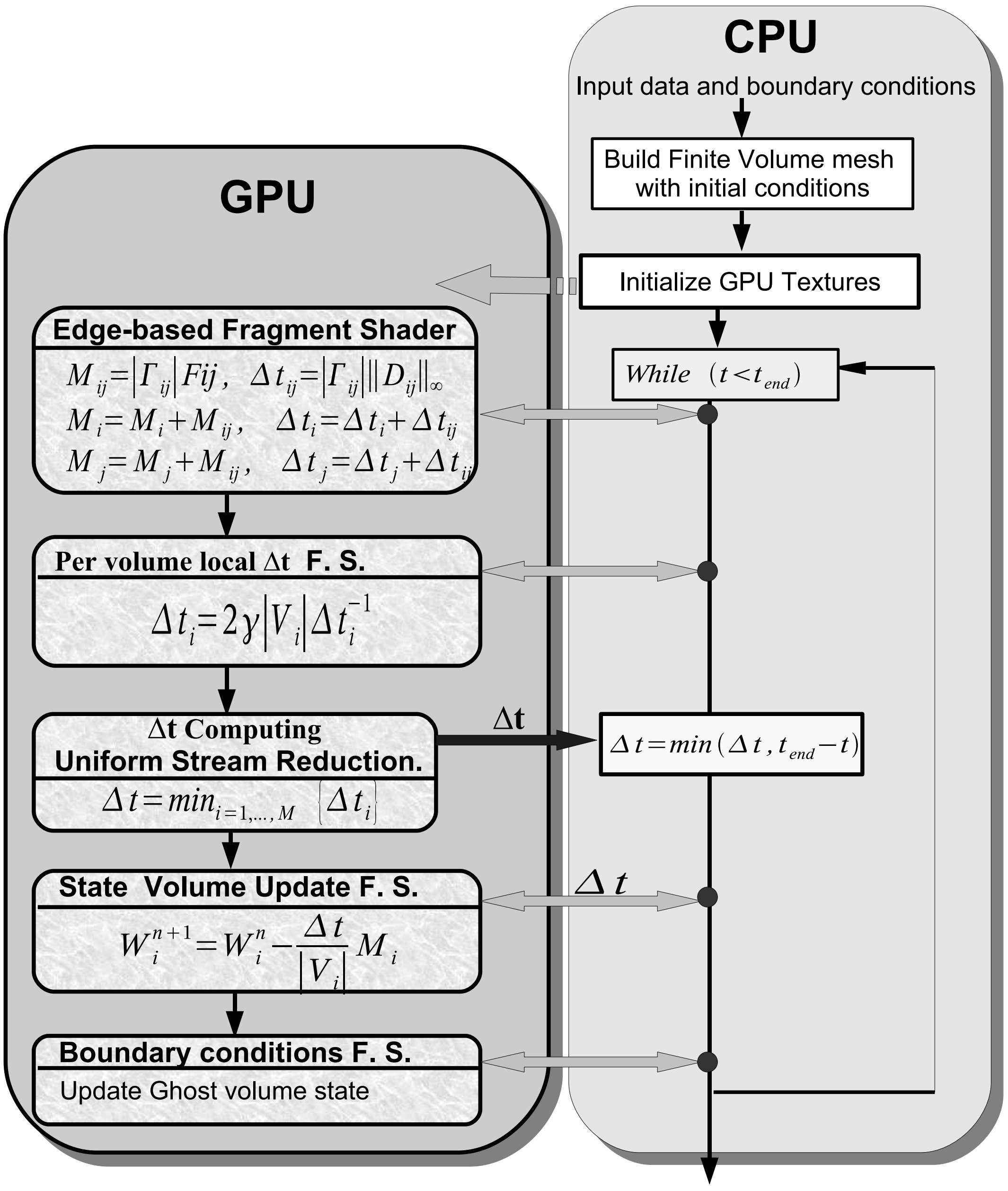


Fig. 7. Computing scheme in a volume-based fragment: for each fragment an instance of the Cg code is run. At each instance other textures can be accessed to obtain input data.



CPU
Input data and boundary conditions
Build Finite Volume mesh with initial conditions
Initialize GPU Textures
$While\ (t<t_{end})$
GPU
Edge-based Fragment Shader
$M_{ij}=|\Gamma_{ij}|Fij,\ \Delta t_{ij}=|\Gamma_{ij}|\|D_{ij}\|_\infty$
$M_i=M_i+M_{ij},\ \Delta t_i=\Delta t_i+\Delta t_{ij}$
$M_j=M_j+M_{ij},\ \Delta t_j=\Delta t_j+\Delta t_{ij}$
Per volume local Δt F. S.
$\Delta t_i=2\gamma|V_i|\Delta t_i^{-1}$
Δt Computing Uniform Stream Reduction.
$\Delta t=min_{i=1,\dots,M}\{\Delta t_i\}$
Δt
$\Delta t=min(\Delta t, t_{end}-t)$
State Volume Update F. S.
$W_i^{n+1}=W_i^n-\frac{\Delta t}{|V_i|}M_i$
Δt
Boundary conditions F. S.
Update Ghost volume state

Fig. 8. CPU-GPU program structure

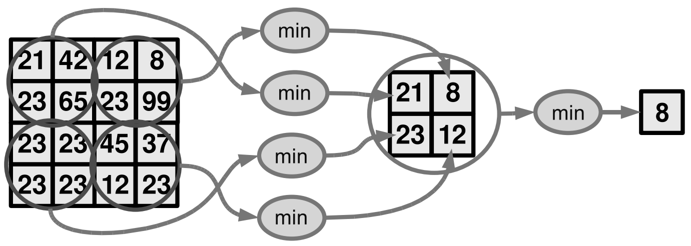


Fig. 9. Stream reduction in GPU

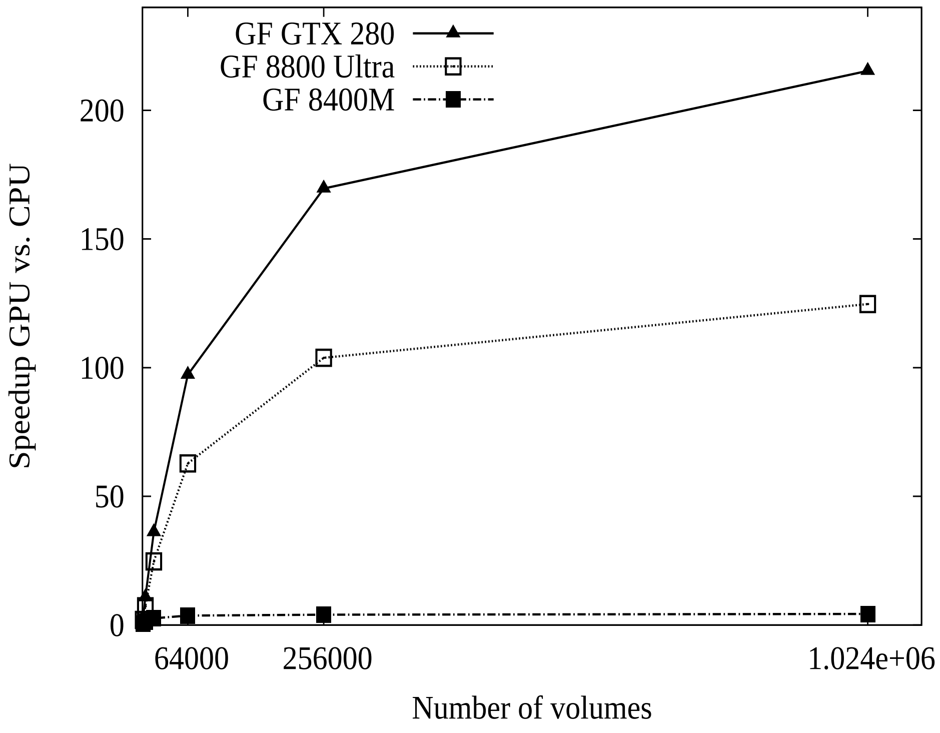


Fig. 10. Test 1. Speedup obtained with respect the CPU code.

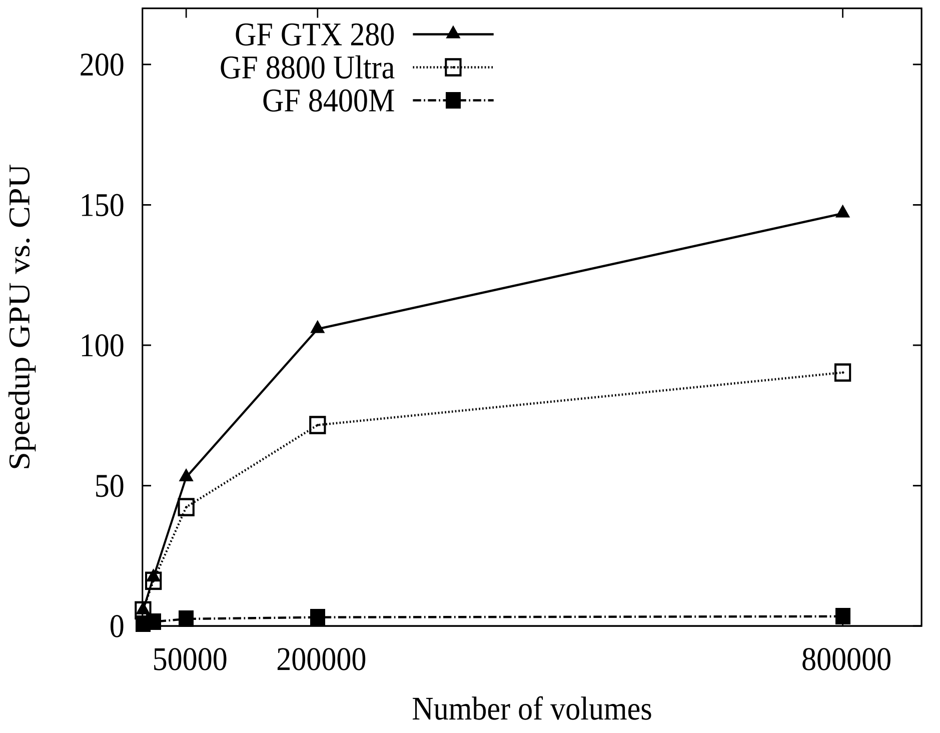
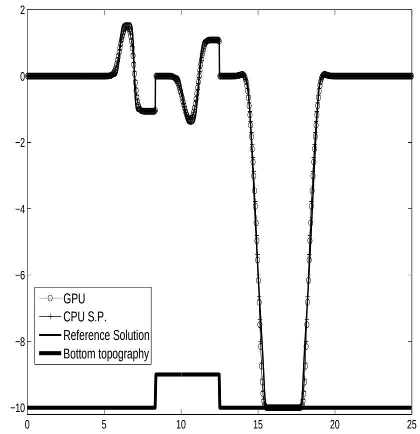
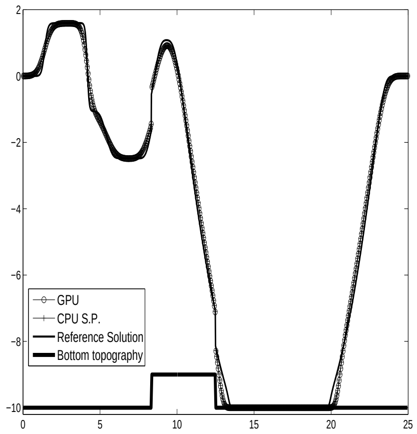


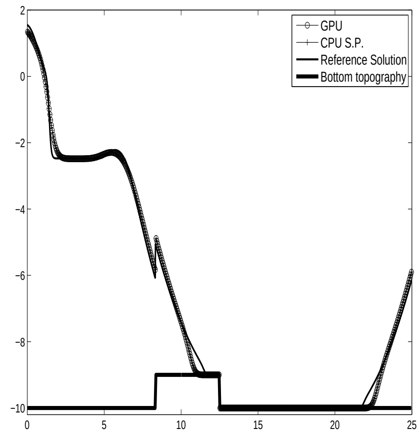
Fig. 11. Test 2. Speedup obtained with respect the CPU code.



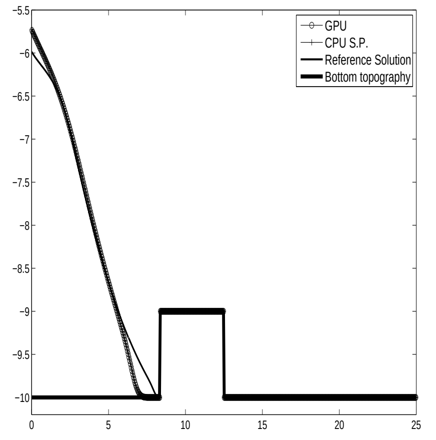
(a) Free Surface at t=0.05 s.



(b) Free Surface at t=0.15 s.

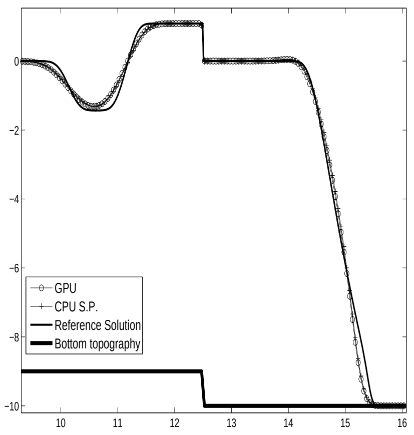


(c) Free Surface at t=0.25 s.

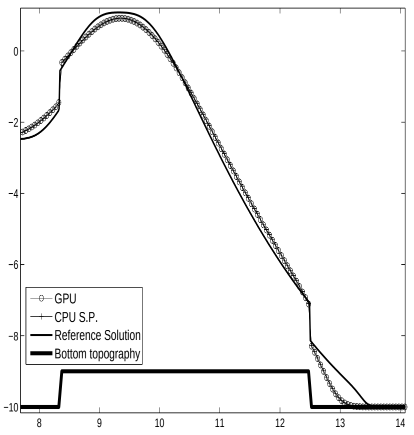


(d) Free Surface at t=0.45 s.

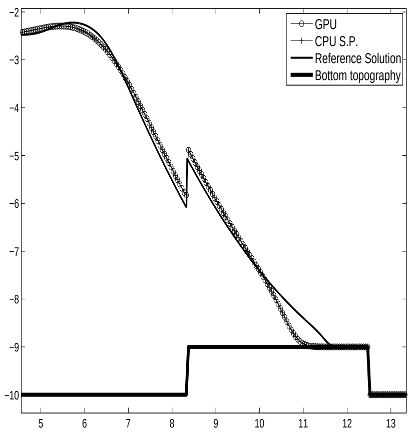
Fig. 12. Test 2: Free surface evolution.



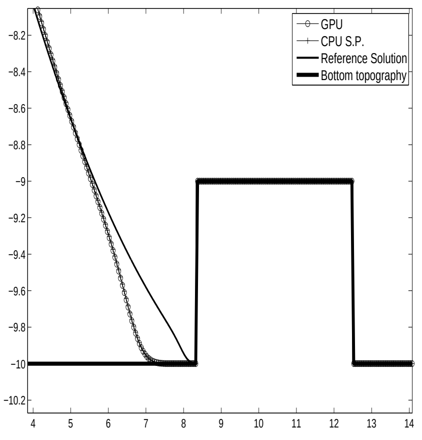
(a) Free Surface at t=0.05 s.



(b) Free Surface at t=0.15 s.



(c) Free Surface at t=0.25 s.



(d) Free Surface at t=0.45 s.

Fig. 13. Test 2: Free surface evolution (zoom)