EINA
Universidad Zaragoza

Máster Universitario en Mecánica Aplicada — *Trabajo Fin de Máster*

2D Shallow Flow Simulation Using GPU Technologies

Asier Lacasta Soto
Grupo de Hidráulica Computacional - alacasta@unizar.es

Directora: Pilar García Navarro[1]

Co-Director: Javier Murillo Castarlenas[1]

(1) Area de Mecánica de Fluidos
Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Curso 2011/2012

Acknowledgements

I would like to express my appreciation to Dra. Pilar García Navarro for her valuable and constructive suggestions during the planning and development of this research work. My grateful thanks are also extended to Dr. Javier Murillo for his help with the model solving my doubts.

I would also like to thank the other members of the group for their advices and points of view when necessary. I would like specially mention to Hector Ratia.

Finally, I wish to thank nVidia for their partial support providing us with a Hardware part under the Nvidia Academic Partnership program.

This work has been developed under project CENIT-TECOAGUA CEN-20091028.

Resumen

Los modelos matemáticos y métodos numéricos implicados en la simulación de flujos con superficie libre han sido estudiados durante tiempo en el Grupo de Hidráulica Computacional de la Universidad de Zaragoza. Estos modelos son la base de nuevos desarrollos como el transporte de sedimento, el modelado de interacción con puentes o el acoplamiento hidrológico. A pesar de la calidad de estos métodos, el coste computacional es muy alto y en gran parte esto se debe a la tecnología numérica que requieren.

Con la finalidad de superar esta limitación, este trabajo estudia la implementación de un código de simulación hidráulica orientada a ejecución en GPU, permitiendo simular un amplio conjunto de situaciones transitorias en gran escala temporal, con un tiempo de simulación razonable.

El coste computacional de éste tipo de herramientas ha sido reducido, tradicionalmente, utilizando técnicas de paralelismo, implicando un alto número de procesadores para reducir el tiempo de cálculo al máximo. En los últimos años, las frecuencias de los procesadores parecen haber alcanzado su límite (Figura 1 extraida de [9]) por lo que las técnicas de paralelismo en procesadores masivos son una nueva opción.

Figure 1: Evolución de las frecuencias de CPU desde 1985 hasta 2011

En este trabajo, se analiza el rendimiento del código implementado en GPU, comparándolo con su equivalente en CPU. Este segudo, viene siendo desarrollado, en su totalidad, en Fortran mientras que el primero, ha sido desarrollado utilizando el lenguaje de programación C, compartiendo el procesamiento geométrico con la versión CPU. Las fucionalidades implementadas en la versión GPU, cubre una gran parte de situaciones de interés, tales como el avance de una inundación, los cambios de fondo y fricción y algunas condiciones de contorno de entrada y de salidas. La implementación del método en GPU no es trivial y requiere de un conocimiento en profundidad del funcionamiento de esta tecnología a bajo nivel. Los beneficios de la versión GPU serán analizados a través de la aceleración repecto a la versión CPU en diferentes tipos de caso.

EL rendimiento del código GPU además, será medido teniendo en cuenta el uso de mallas no estructuradas, las cuales suelen ser necesarias en muchos codigos de CFD. Para su simulación, se utilizará la GPU Tesla c2075 de nVidia. Además se utilizará el estándar CUDA, que hace la programación más sencilla que otros estándar en programción GPU, permietiendo al programador exprimir los beneficios de esta tecnología.

Abstract

The mathematical models and numerical methods implied in the resolution of free surface flows have been studied for a long time within the Computational Hydraulic Group at the Universidad Zaragoza. They support new developments such as sediment transport, bridges modeling or hydrological coupling. Despite the quality that the numerical solvers proposed by the group offer, the computational cost of these methods is very high, due to the complexity of the numerical tools required.

In order to avoid this limitation, the present work studies the implementation of a scientific hydraulic simulation tool oriented to be run on GPU, allowing to simulate a wide range of situations over large time scale problems, that otherwise can not be computed at an affordable cost.

The computational cost has been traditionally reduced by using parallel techniques, involving a large number of processors in order to reduce the simulation time as much as possible. Since CPU frequencies seem to be reaching their maximum capacity (Figure 2 extracted from [9]), nowadays Many-Core parallel techniques appear to be an interesting option.

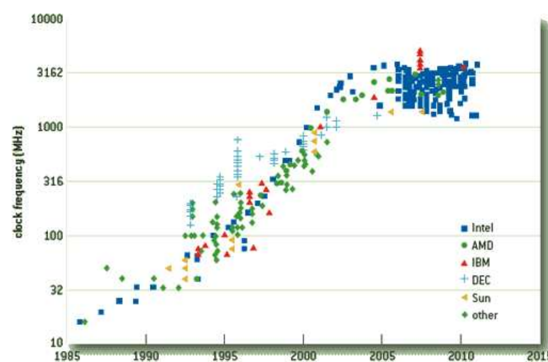


Figure 2: CPU Frequency evolution since 1985 until 2011

The performance of the GPU version is analyzed comparing both CPU and GPU versions of the same code. While the former was fully developed in Fortran language, the numerical

kernel of the new GPU version has been written in C, sharing the geometrical preprocessing module with the CPU version. The functionalities implemented in the GPU version cover a wide range of situations as they include all the characteristics that are desirable in the context of shallow flow simulation: flooding advance, friction and bed slope source-terms as weel as inlet and outlet boundary conditions. The implementation of these requirements in the context of realistic simulations is not straightforward. This is explained when considering that, contrary to other programming languages, the GPU version requires a good comprehension of the low level operations, that does not allow a direct conventional implementation. The benefits of the GPU version will be analyzed in depth focusing on speed-up gain in complex cases.

The performance of the GPU code is analyzed in depth to ensure not only the efficiency but also the possibilities of GPU programming when using unstructured meshes, that are often required in CFD codes. A Tesla c2075 nVidia GPU has been used in the present study. Moreover, it has been developed using nVidia-CUDA standard, which makes friendly the programming for general purpose applications, allowing the programmer to exploit the many-core paradigm.

Contents

1 Introduction **1**
1.1 Context and assumptions . . . 1
1.2 Structure of the report . . . 2

2 Mathematical Model and Numerical Method **3**
2.1 Approximate Riemann solution . . . 3
2.2 Application to the 2D Shallow Water equations . . . 6
2.3 Numerical resolution . . . 7

3 CUDA Technology Overview **11**
3.1 GPU Technology history . . . 11
3.2 nVidia CUDA technology . . . 12
3.3 CUDA development . . . 13
3.3.1 Example of implementation in a 1D case . . . 14
3.4 Results *All that glitters is not gold* . . . 16

4 Implementation **19**
4.1 Model overview . . . 19
4.2 Memory coalescing . . . 21
4.3 Gathering data avoiding bottleneck . . . 23
4.4 Writing output files . . . 24
4.5 Compilation and other issues . . . 26

5 Results **29**
5.1 Precision: A test-case with analytical solution . . . 29
5.2 Performance: A large-scale simulation at Júcar River . . . 33
5.3 Comparing with a distributed memory parallel implementation . . . 39

6 Conclusions and future work **45**

Bibliography **47**

List of Figures

1 Evolución de las frecuencias de CPU desde 1985 hasta 2011 iii
2 CPU Frequency evolution since 1985 until 2011 v

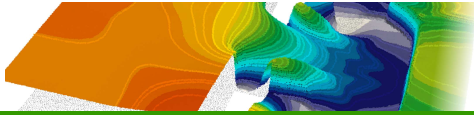
2.1 Riemann problem in 2D along the normal direction to a cell side. 5

3.1 `thread, block, grid` scheme composition 12
3.2 Description of our Fermi c2075 GPU based on GF100/GF110 Architecture. . . . 13
3.3 Execution pipeline for a Strem Multiprocessor (left) which process block number 4 (right) 13

4.1 Execution trace and performance detail for a time-step using Paraver 20
4.2 Structured mesh with Cell Numbering detail (Right) and Wall Numbering detail (Left) example 21
4.3 Misaligned and Coalesced access pattern to compute the flux variation for any group of elements following the scheme of Right, Left, Down, Up for W data (Stored by cell) in a mesh ordered as Figure 4.2. Light coloured correspond to the processed element 5, wich implies cells 2, 4, 6 and 8. 22
4.4 Unstructured mesh with Cell Numbering detail (Right) and Wall Numbering detail (Left) example 22
4.5 Uncoalesced access pattern to get W data (Stored by cell). Processing wall 9 is light coloured when it accesses to cell 8 (i=9, c1=8) 23
4.6 Gathering minimum Δt for all the domain 24
4.7 Asynchronus dumping data diagram. 25
4.8 Flux diagram for the application. Green-highlighted is the ported slice of the code 27

5.1 Left: Bed level and initial water depth state for test case 1. 29
5.2 Test case 1. Left: GPU Simulated results for h and Right: CPU Simulated results for h at $t = 42.03s$. 30
5.3 Test case 1. Left: GPU Simulated result for $|v|$ and Right: CPU Simulated results for $|v|$ at $t = 42.03s$. 30

5.4 Left: Initial state h_0 for the conflictive cell. Center: h^1 for CPU. ϵ accuracy involves wall treatment as solid edge implying an increasing in it water depth. Right: h^1 for GPU. ϵ accuracy involves wall treatment as non solid edge so that water level decrease at cell i and increase at cell j. 31
5.5 From Left to right, Top to down, h+z for t=T/4, T/2, 3T/4 and T 32
5.6 From Left to right, Top to down, u for t=T/4, T/2, 3T/4 and T 32
5.7 From Left to right, Top to down, v for t=T/4, T/2, 3T/4 and T 33
5.8 From Left to right, Top to down, h for t=0, T/4, T/2, 3T/4 and T 34
5.9 Left: Sumácarcel photography. Right: simulation mesh 35
5.10 Water depth evolution for (Left-right, Top-down) $t = 5, 10, 15, 20, 25, 30h$ 35
5.11 Gauges position 36
5.12 Simulated and estimated water depth in 1-11 Gauges. 40
5.13 Simulated and estimated water depth in 12-21 Gauges. 41
5.14 Tous synthetic hydrograph for D_1 (Right) and D_2 (Left) 42
5.15 Comparison of Left: Coarse mesh velocity module and Righ: Refined mesh velocity module at $t = 13h$ 42
5.16 Initial conditions of water depth and mesh plot 42
5.17 5-0 Dam-Break simulation for (Right-Left, Top-Down) t=5, 10, 15, 20, 25, 30 seconds 43



1

Introduction

The present work deals with of the efficient implementation of a scientific purpose code oriented to make hydraulic simulations that require a very high computational load. These calculations could range from a dam break simulation to the consequences of a river flooding.

The code is based on a numerical resolution of the shallow water model used to simulate water fluxes under certain hypothesis. Free surface fluxes of interest to Hydraulic Engineering are usually formulated under the shallow water model which assumes that vertical lengths are lower than horizontal scales in the problem. The depth averaged system of equations resulting from this approach allows to make a temporal description of the flow field as a function of water depth and horizontal velocity components u, v in x and y axis respectively.

The governing system of partial differential equations is hyperbolic and, in general, does not have exact solution. Therefore, numerical methods are required to reach the solution or to approximate it. The question of what is the most suitable method to solve it is still open but finite volume schemes are widely used.

1.1 Context and assumptions

The Computational Hydraulics Group at the University of Zaragoza (http://ghc.unizar.es) is involved with both research and teaching activities related to the topic of this project. This research team has been working on Computational Hydraulic Research since 1986. The results have been published in many international journals and have led to actual knowledge transfer models that are nowadays used by private and public bodies in Spain. The numerical models of free surface flows developed by this research team has led to efficient, robust and accurate simulation software tools. The research team has extended the numerical schemes making feasible the application to realistic cases found in engineering applications, where the importance of the source terms in the equations, mainly related with the bathymetry of the bed in river flows, requires special numerical treatments. In order to involve all possible scenarios, two different modelling lines have been explored. A one-dimensional research line to analyse rivers and channels, and a two-dimensional research line, where the transversal component of the flow is of importance, able

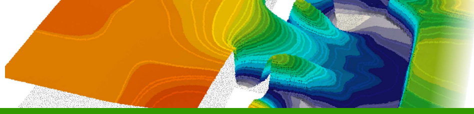
to handle more complex situations. This approach may lead to very time consuming simulations.

To study the performance of the GPU version, it has been compared to the CPU version. That has been developed for a long time. The numerical kernel in the GPU version has been written in C, sharing the geometrical preprocess with the CPU version. Although the CPU version has several functionalities implemented, the GPU version covers only a few of them. In particular, the Shallow-Water equations discretization using Roe solver including wet/dry boundaries, friction source-term, and two inlet and outlet boundaries. With this implementation, the gain of the GPU version will be studied.

Both the CPU and GPU versions work with the same data-structures. Furthermore, the numerical kernel in both versions is optimized so that they to make more or less the same number of operations and are compiled with the same options in order to apply a correct analysis for the comparison.

1.2 Structure of the report

The report has been structured in 5 sections. First the mathematical model and numerical scheme used to solve the free surface flow equations are introduced. Second one describes the way to program a general numerical solver in GPU's, using as example the 1D transport equation. Furthermore, in this second part the hardware composition of the GPU and the CUDA model to develop to it are also described. The third part explains the main problems found in the implementation of the model. These problems are explained as a general way to solve problems related to the numerical solvers. The fourth part contains three test cases where accuracy and performance are studied comparing with both, serial and parallel implementations of the method. The last part describes our conclusions as well as the desirable future work improvements.



2 Mathematical Model and Numerical Method

We are interested in the simulation of a problem that can be formulated as a system of conservation laws with source term as follows

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} + \frac{\partial \mathbf{G}(\mathbf{U})}{\partial y} = \mathbf{S}(\mathbf{U}, x, y) \tag{2.1}$$

System (2.1) is time dependent and non linear. Under the hypothesis of dominant advection, it can be classified and numerically dealt with as belonging to the family of hyperbolic systems. It includes the existence of a Jacobian matrix of the flux normal to a direction given by the unit vector $\mathbf{n}$, $\mathbf{E} \cdot \mathbf{n}$. Defining $\mathbf{E} \cdot \mathbf{n} = \mathbf{F} n_x + \mathbf{G} n_y$, the Jacobian can be written as

$$\mathbf{J_n} = \frac{\partial \mathbf{E} \cdot \mathbf{n}}{\partial \mathbf{U}} = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} n_x + \frac{\partial \mathbf{G}}{\partial \mathbf{U}} n_y \tag{2.2}$$

The Jacobian can be used to form de basis of the upwind numerical discretization.

2.1 Approximate Riemann solution

The previous differential formulation can be reinterpreted over a volume (or grid cell) Ω using the integral formulation as follows

$$\frac{\partial}{\partial t} \int_\Omega \mathbf{U} d\Omega + \int_\Omega (\vec{\nabla} \mathbf{E}) d\Omega = \int_\Omega \mathbf{S} d\Omega \tag{2.3}$$

which becomes, using the Gauss theorem

$$\frac{\partial}{\partial t} \int_\Omega \mathbf{U} d\Omega + \oint_{\partial\Omega} \mathbf{E} \cdot \mathbf{n} dl = \int_\Omega \mathbf{S} d\Omega \tag{2.4}$$

where $\mathbf{n} = (n_x, n_y)$ is the outward unit normal vector to the volume Ω.

Considering the complete spatial domain discretized in computational cells Ω_i and using the conventional cell-average notation, the solution $\mathbf{U_i^n}$ inside the cell for $\mathbf{U}(x, y, t)$

$$\mathbf{U}_i^n = \frac{1}{A_i} \int_{\Omega_i} \mathbf{U}(x, y, t^n) d\Omega \tag{2.5}$$

being A_i the cell area. Assuming a piecewise representation of the conserved variables, (2.4) could be written as

$$\frac{\partial}{\partial t}\int_{\Omega_i} \mathbf{U} d\Omega + \sum_{k=1}^{NE} \mathbf{E}_j \cdot \mathbf{n}_k l_k = \int_{\Omega_i} \mathbf{S} d\Omega \tag{2.6}$$

where $\mathbf{E}_j$ is the value of the function $\mathbf{E}$ at the neighbouring cell j connected through the edge k, $\mathbf{n}_k$ is the outward unit normal vector to the cell edge k, l_k is the corresponding edge length and NE is the number of edges around cell i. Considering the quantity $\mathbf{E}_i$ uniform per cell i and that

$$\sum_{k=1}^{NE} \mathbf{n}_k l_k = 0 \tag{2.7}$$

equation (2.6) is written as

$$\frac{\partial}{\partial t}\int_{\Omega_i} \mathbf{U} d\Omega + \sum_{k=1}^{NE} (\delta\mathbf{E})_k \cdot \mathbf{n}_k l_k = \int_{\Omega_i} \mathbf{S} d\Omega \tag{2.8}$$

with $\delta\mathbf{E} = \mathbf{E}_j - \mathbf{E}_i$.

In the Roe approach [24], the solution of each RP is obtained from the exact solution of a locally linearized problem. In the 2D framework the solution is obtained reducing each RP at each k edge to a 1D Riemann problem projected onto the direction of $\mathbf{n}$. The linearized solution must fulfill the Consistency Condition. In the 2D case the integral of the approximate solution $\hat{\mathbf{U}}(x', t)$ of the k linearized RP over a suitable control volume must be equal to the integral of the exact solution $\mathbf{U}(x', t)$ over the same control volume, with x' the coordinate normal to the cell edge k, Figure 2.1. Then in each k Riemann problem with initial values $\mathbf{U}_i, \mathbf{U}_j$, in a time interval $[0, 1]$ and a space interval $[-X', X']$, where

$$-X' \le \lambda_{min}, \quad X' \ge \lambda_{max} \tag{2.9}$$

and $\lambda_{min}, \lambda_{max}$ the positions of the slowest and the fastest wave at $t = 1$, in a k egde, the solution $\hat{\mathbf{U}}(x', 1)$ at time $t = 1$ must satisfy the following property:

$$\int_{-X'}^{+X'} \hat{\mathbf{U}}(x', 1)\, dx' = \int_{-X'}^{+X'} \mathbf{U}(x', 1)\, dx' \tag{2.10}$$

so using (2.8) the Consistency Condition becomes:

$$\int_{-X'}^{+X'} \hat{\mathbf{U}}(x', 1)\, dx' = X'(\mathbf{U}_i + \mathbf{U}_j) - \delta\mathbf{E}_k \cdot \mathbf{n}_k + \int_0^1 \int_{-X'}^{+X'} \mathbf{S}\, dx'\, dt \tag{2.11}$$

Since the source terms are not necessarily constant in time, we assume the following time linearization of the Consistency Condition:

$$\int_{-X'}^{+X'} \hat{\mathbf{U}}(x', 1)\, dx' = X(\mathbf{U}_i + \mathbf{U}_j) - (\delta\mathbf{E} - \mathbf{T})_k \mathbf{n}_k \tag{2.12}$$

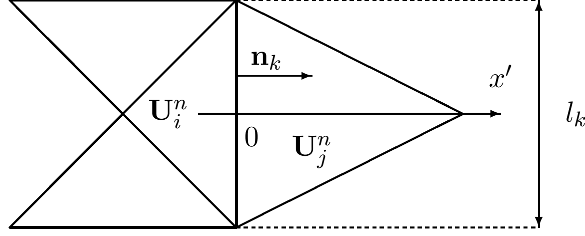


Figure 2.1: Riemann problem in 2D along the normal direction to a cell side.

where following previous work, [28]

$$\int_{-X'}^{+X'} \mathbf{S}(x',0)\, dx' = (\mathbf{Tn})_k^n \tag{2.13}$$

where $\mathbf{T}$ is a suitable numerical source matrix. This enables the following formulation of (2.8)

$$\frac{\partial}{\partial t}\int_{\Omega} \mathbf{U} d\Omega_i + \sum_{k=1}^{NE} (\delta \mathbf{E} - \mathbf{T})_k \mathbf{n}_k l_k = 0 \tag{2.14}$$

that is approximated by using the following linear problem

$$\begin{aligned} &\tfrac{\partial}{\partial t}\textstyle\int_{\Omega} \hat{\mathbf{U}} d\Omega_i + \sum_{k=1}^{NE} \mathbf{J}_{\mathbf{n},k}^{*} \delta \hat{\mathbf{U}}_k l_k = 0 \\ &\hat{\mathbf{U}}(x',0)_k = \begin{cases} \mathbf{U}_i & if \quad x' < 0 \\ \mathbf{U}_j & if \quad x' > 0 \end{cases} \end{aligned} \tag{2.15}$$

Integrating 2.15 over the same control volume as before the following expression is obtained for each k edge

$$\int_{-X'}^{+X'} \hat{\mathbf{U}}(x',1)\, dx' = X\,(\mathbf{U}_i + \mathbf{U}_j) - \mathbf{J}^*\,(\mathbf{U}_j - \mathbf{U}_i) \tag{2.16}$$

and since we want to satisfy (2.12), the constraint that follows is:

$$(\delta \mathbf{E} - \mathbf{T})_k \mathbf{n}_k = \tilde{\mathbf{J}}^*\,(\mathbf{U}_j - \mathbf{U}_i) \tag{2.17}$$

Due to the non-linear character of the flux matrix $\mathbf{E}$, the definition of an approximated Jacobian matrix, $\widetilde{\mathbf{J}}_{\mathbf{n},k}$, allows for a local linearization

$$\delta(\mathbf{En})_k = \widetilde{\mathbf{J}}_{\mathbf{n},k} \delta \mathbf{U}_k \tag{2.18}$$

and is exploited here [24]. This approach provides a set of three real eigenvalues $\widetilde{\lambda}_k^m$ and eigenvectors $\widetilde{\mathbf{e}}_k^m$. Then, it is possible to define two matrices $\widetilde{\mathbf{P}} = (\widetilde{\mathbf{e}}^1, \widetilde{\mathbf{e}}^2, \widetilde{\mathbf{e}}^3)$ and $\widetilde{\mathbf{P}}^{-1}$ with the following property

$$\widetilde{\mathbf{J}}_{\mathbf{n},k} = \widetilde{\mathbf{P}}_k \widetilde{\mathbf{\Lambda}}_k \widetilde{\mathbf{P}}_k^{-1} \tag{2.19}$$

The difference in vector $\mathbf{U}$ across the grid edge and the source term are projected onto the matrix eigenvectors basis:

$$\delta \mathbf{U}_k = \widetilde{\mathbf{P}}_k \mathbf{A}_k \qquad (\mathbf{Tn})_k = \widetilde{\mathbf{P}}_k \mathbf{B}_k \tag{2.20}$$

with $\mathbf{A}_k = \begin{pmatrix} \alpha^1 & \alpha^2 & \alpha^3 \end{pmatrix}_k^T$ and $\mathbf{B}_k = \begin{pmatrix} \beta^1 & \beta^2 & \beta^3 \end{pmatrix}_k^T$. Expressing all terms more compactly:

$$\delta(\mathbf{E}\cdot\mathbf{n})_k - (\mathbf{T}\cdot\mathbf{n})_k = \sum_{m=1}^{N_\lambda} \left(\widetilde{\lambda}\,\theta\alpha\widetilde{\mathbf{e}}\right)_k^m \tag{2.21}$$

with

$$\theta_k^m = \left(1 - \frac{\beta}{\widetilde{\lambda}\alpha}\right)_k^m \tag{2.22}$$

Finally, it is possible to define the desired matrix in (2.17)

$$\widetilde{\mathbf{J}}_k^* = (\widetilde{\mathbf{P}}\widetilde{\mathbf{\Lambda}}^*\widetilde{\mathbf{P}}^{-1})_k \tag{2.23}$$

with $\widetilde{\mathbf{\Lambda}}^* = \widetilde{\mathbf{\Lambda}}\mathbf{\Theta}$, where $\widetilde{\mathbf{\Lambda}}_k$ is a diagonal matrix with eigenvalues $\widetilde{\lambda}_k^{m,*}$ in the main diagonal and $\mathbf{\Theta}_k$ is a diagonal matrix with θ_k^m in the main diagonal:

$$\widetilde{\mathbf{\Lambda}}_k = \begin{pmatrix} \widetilde{\lambda}^1 & 0 & 0 \\ 0 & \widetilde{\lambda}^2 & 0 \\ 0 & 0 & \widetilde{\lambda}^3 \end{pmatrix}_k \qquad \Theta_k = \begin{pmatrix} \theta^1 & 0 & 0 \\ 0 & \theta^2 & 0 \\ 0 & 0 & \theta^3 \end{pmatrix}_k \tag{2.24}$$

2.2 Application to the 2D Shallow Water equations

The two-dimensional shallow water equations, which represent depth averaged mass and momentum conservation, can be obtained from the Navier-Stokes equations. Neglecting diffusion of momentum due to viscosity and turbulence, wind effects and the Coriolis term, they form a system of equations [2] as in (2.1), where

$$\mathbf{U} = (h,\ q_x,\ q_y)^T \tag{2.25}$$

are the conserved variables with h representing the water depth, $q_x = hu$ and $q_y = hv$, with (u, v) the depth averaged components of the velocity vector $\mathbf{u}$ along the (x, y) coordinates respectively. The fluxes of these variables are given by:

$$\mathbf{F} = \left(q_x,\ \frac{q_x^2}{h} + \frac{1}{2}gh^2,\ \frac{q_x q_y}{h}\right)^T, \qquad \mathbf{G} = \left(q_y,\ \frac{q_x q_y}{h}\ \frac{q_y^2}{h} + \frac{1}{2}gh^2\right)^T \tag{2.26}$$

where g is the acceleration of the gravity. The source terms of the system are the bed slope and the friction terms:

$$\mathbf{S} = \left(0,\ \frac{p_{b,x}}{\rho_w} - \frac{\tau_{b,x}}{\rho_w},\ \frac{p_{b,y}}{\rho_w} - \frac{\tau_{b,y}}{\rho_w}\right)^T \tag{2.27}$$

where the bed slopes of the bottom level z are

$$\frac{p_{b,x}}{\rho_w} = -gh\frac{\partial z}{\partial x}, \qquad \frac{p_{b,y}}{\rho_w} = -gh\frac{\partial z}{\partial y} \tag{2.28}$$

and the friction losses are written in terms of the Manning's roughness coefficient n:

$$\frac{\tau_{b,x}}{\rho_w} = ghS_{fx} \quad S_{fx} = \frac{n^2 u\sqrt{u^2+v^2}}{h^{4/3}}, \qquad \frac{\tau_{b,y}}{\rho_w} = ghS_{fy} \quad S_{fy} = \frac{n^2 v\sqrt{u^2+v^2}}{h^{4/3}} \tag{2.29}$$

2.3 Numerical resolution

Following Godunov's method, the solutions of the RP's are evolved for a time equal to the time step and the resulting solution is cell-averaged. The volume integral in the cell at time t^{n+1} leads to the updating numerical scheme as:

$$\mathbf{U}_i^{n+1}A_i = \mathbf{U}_i^n A_i - \sum_{k=1}^{NE}\sum_{m=1}^{3}(\widetilde{\lambda}^- \theta\alpha\widetilde{\mathbf{e}})_k^m l_k \Delta t \tag{2.30}$$

with $\widetilde{\lambda}_k^{\pm,m} = \frac{1}{2}(\widetilde{\lambda} \pm |\widetilde{\lambda}|)_k^m$.

When applied to the shallow water system presented in section 2.2 the approximate Jacobian $\widetilde{\mathbf{J}}_{\mathbf{n},k}$ for the homogeneous part is constructed with the following averaged variables [24]

$$\widetilde{u}_k = \frac{u_i\sqrt{h_i} + u_j\sqrt{h_j}}{\sqrt{h_i}+\sqrt{h_j}}, \qquad \widetilde{v}_k = \frac{v_i\sqrt{h_i} + v_j\sqrt{h_j}}{\sqrt{h_i}+\sqrt{h_j}}, \qquad \widetilde{c}_k = \sqrt{g\frac{h_i+h_j}{2}} \tag{2.31}$$

leading to

$$\widetilde{\lambda}_k^1 = (\widetilde{\mathbf{u}}\mathbf{n} - \widetilde{c})_k, \qquad \widetilde{\lambda}_k^2 = (\widetilde{\mathbf{u}}\mathbf{n})_k, \qquad \widetilde{\lambda}_k^3 = (\widetilde{\mathbf{u}}\mathbf{n} + \widetilde{c})_k \tag{2.32}$$

and

$$\widetilde{\mathbf{e}}_k^1 = \begin{pmatrix} 1 \\ \widetilde{u} - \widetilde{c}n_x \\ \widetilde{v} - \widetilde{c}n_y \end{pmatrix}_k, \qquad \widetilde{\mathbf{e}}_k^2 = \begin{pmatrix} 0 \\ -\widetilde{c}n_y \\ \widetilde{c}n_x \end{pmatrix}_k, \qquad \widetilde{\mathbf{e}}_k^3 = \begin{pmatrix} 1 \\ \widetilde{u} + \widetilde{c}n_x \\ \widetilde{v} + \widetilde{c}n_y \end{pmatrix}_k \tag{2.33}$$

When cell averaging the solution in the 1D dimensional case the time step Δt is taken small enough so that there is no interaction of waves from neighbouring Riemann problems, attending to a distance $\Delta x/2$. In the 2D framework, considering unstructured meshes, the equivalent distance to Δx, that will be referred to as χ_i in each cell i must consider the volume of the cell and the length of the shared k edges.

$$\chi_i = \frac{A_i}{\max_{k=1,NE} l_k} \tag{2.34}$$

Considering that each k RP is used to deliver information between each pair of neighbouring cells of different size, the associated distance $\min(A_i, A_j)/l_k$ is relevant, so in case that $\hat{h}(x', t) \geq 0$ in all k RP's the time step is limited by

$$\Delta t \leq CFL\ \Delta t^{\widetilde{\lambda}} \qquad \Delta t^{\widetilde{\lambda}} = \frac{\min(\chi_i, \chi_j)}{\max_{m=1,2,3}|\widetilde{\lambda}^m|} \tag{2.35}$$

The previous stability condition is insufficient in presence of relatively important source terms. The systematic control of numerical stability in those cases has been a matter of recent research in the group as it is related with the applicability of the scheme to real situations. A simple generalization of the CFL condition paying attention to the existence of the source terms can lead to extremely small values of Δt various orders of magnitude smaller than the value dictated by the homogeneous condition, hence rendering the method impractical. This can be avoided by means of a reconstruction of the approximate solution $\hat{\mathbf{U}}(x',t)$ that is not detailed here for the sake of conciseness. The strategy proposed is based on enforcing positive values of auxiliary quantities h_i^*

$$h_i^* = h_i^n + \alpha_k^1 - \left(\frac{\beta}{\widetilde{\lambda}}\right)_k^1 \geq 0 \tag{2.36}$$

and h_j^{***}

$$h_j^{***} = h_j^n - \alpha_k^3 + \left(\frac{\beta}{\widetilde{\lambda}}\right)_k^3 \geq 0 \tag{2.37}$$

so that, when they become negative, the numerical source term is reduced instead of reducing the time step size. For more details, see [21, 18].

Furthermore, following the unified discretization in [6] the non-conservative term $(\mathbf{Tn})_k$ in (2.13) at a cell edge is written [20] as:

$$(\mathbf{Tn})_k = \begin{pmatrix} 0 \\ \left(\frac{p_b}{\rho_w} - \frac{\tau_b}{\rho_w}\right) n_x \\ \left(\frac{p_b}{\rho_w} - \frac{\tau_b}{\rho_w}\right) n_y \end{pmatrix}_k \tag{2.38}$$

where $\frac{p_b}{\rho_w}$ and $\frac{\tau_b}{\rho_w}$ attend to the pressure and friction exerted on the bed respectively.

In this work the following expression for the thrust term $\frac{p_b}{\rho_w}$ is proposed:

$$\left(\frac{p_b}{\rho_w}\right)_k = \begin{cases} \max\left(\left(\frac{p_b}{\rho_w}\right)^a, \left(\frac{p_b}{\rho_w}\right)^b\right)_k & \text{if } \delta d\ \delta z \geq 0 \text{ and } (\widetilde{\mathbf{u}}\mathbf{n})\delta z > 0 \\ \left(\frac{p_b}{\rho_w}\right)_k^b & \text{otherwise} \end{cases} \tag{2.39}$$

where $d = (h + z)$ and

$$\left(\frac{p_b}{\rho_w}\right)_k^a = -g(\widetilde{h\delta z})_k \qquad \left(\frac{p_b}{\rho_w}\right)_k^b = -g\left(h_r - \frac{|\delta z'|}{2}\right)\delta z' \tag{2.40}$$

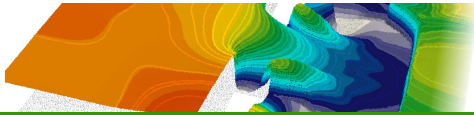
with

$$r = \begin{cases} i & \text{if } \delta z \geq 0 \\ j & \text{if } \delta z < 0 \end{cases} \qquad \delta z' = \begin{cases} h_i & \text{if } \delta z \geq 0 \text{ and } d_i < z_j \\ h_j & \text{if } \delta z < 0 \text{ and } d_j < z_i \\ \delta z & \text{otherwise} \end{cases} \tag{2.41}$$

The discretization of the friction term based on [21] is applied

$$\left(\frac{\tau_b}{\rho_w}\right)_k = g(\widetilde{hS_f})_k d_{\mathbf{n}} \qquad S_{f,k} = \left(\frac{n^2 \widetilde{\mathbf{u}}\mathbf{n}|\widetilde{\mathbf{u}}|}{\max(h_i, h_j)^{4/3}}\right)_k \tag{2.42}$$

with d_n the normal distance between neighbor cell centers.



3

CUDA Technology Overview

Nowadays, GPU technologies start to conquer from ordinary business applications to sciencitific applications. This *general purpose* orientation is denomined GPGPU[1], allowing its developers to reach higher performance than in coventional architectures (Single Instruction Single Data) where the operations are currecntly performed sequentially. In the case of scientific computation, the GPGPU paradigm performs the numerical methods.

nVidia has been working in the improvement of the GPGPU paradigm, creating the CUDA toolkit. CUDA toolkit is a parallel architecture for graphic processing which implements an intruction-set oriented to the GPU memory access and operations in C. Other more general implementations have been performed through open-source platforms such as OpenCL and others like PGI-Cuda as propietary-source. OpenCL has the main advantage of being hardware-independent. It implies that the same code could be executed on both nVidia and ATI GPUs. The main disadvantage is that the learning-curve is harder than for the CUDA toolkit. The other option is PGI-Cuda. It has the main advantage in the support of CUDA primitives for Fortran but the disadvantage is the cost of it. So, as we are interested in simulating at nVidia GPUs, the implementation of the code has been developed using CUDA-Toolkit.

3.1 GPU Technology history

Since the advent of OpenGL, GPUs added programmable shading to their capabilities. Each pixel could incoporate its processing as a program to be shown on screen after applying it. nVidia was the first to produce a chip capable of programmable shading. In 2002, ATI developed the first Direct3D 9.0 accelerator, which implemented looping and lengthy floating point math, becoming as flexible as CPU and orders of magnitude faster for image-array operations.

Abstracting the graphical purpose and taking a double-point array as if it were a vertex-array, the same operations were able to be applied, so with the nVidia CUDA Toolkit, a new programming model for GPU computing was stablished. After its appearance, OpenCL became broadly supported allowing developers coding for AMD/ATI GPUs.

[1]General Purpose Graphic Processor Unit

3.2 nVidia CUDA technology

The present work has been developed using an nVidia Tesla GPU. The particular organization and how it works is explained below and has followed [11]. Most of the details are common with the previous GPU generations and it is previsible that will be common with future generations too.

There are two main points of view when explaining how CUDA works. The first is based on the hardware architecture. The minimum unit is the Streaming Processor (SP), where a single thread is executed. A group of SP's form the Streaming Multiprocessor (SM), tipically with 32 SP's. Finally, a GPU is composed by between 2 and 16 SM's. The second point of view is based on the way CUDA applications are developed. The minimum unit is called Thread. Threads are identified by labels ranging between 0 and `blockDim`. The group of Threads is called Block, and it contains a (recommended) 32 multiple number of Threads. Finally any group of Blocks is called Grid. These elements are illustrated on Figure 3.1.

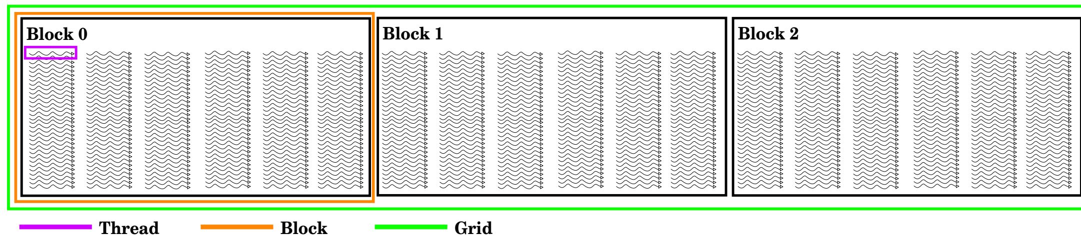


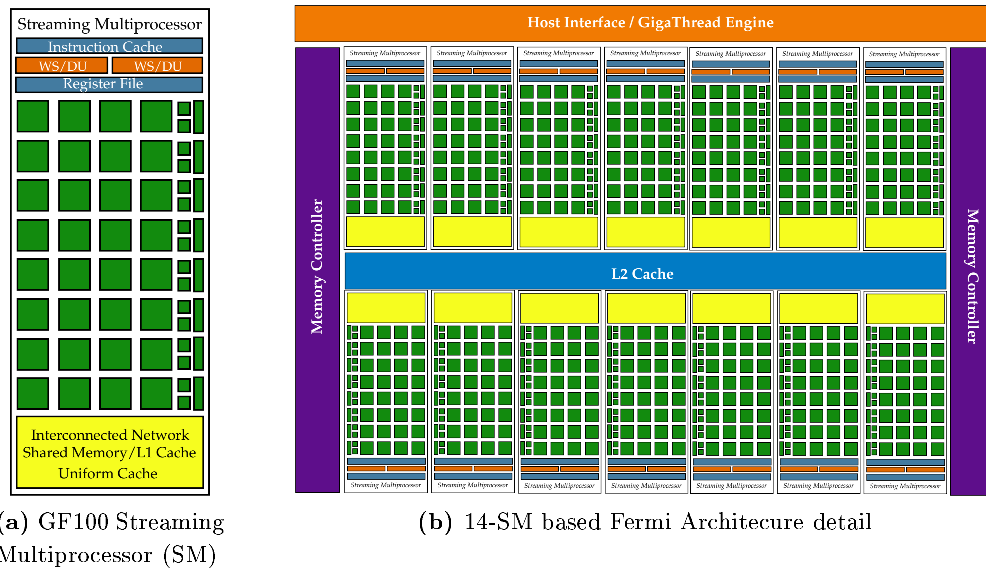
Figure 3.1: `thread, block, grid` scheme composition

Actual nVidia GPU's performs the threads scheduling inside the SM in groups of 32 called Warps (we also recommend [15] for future considerations). Each SM features two Warp schedulers and two instruction dispatch units, allowing two Warps to be issued and executed concurrently. Fermi's dual Warp scheduler selects two Warps, and issues one instruction from each Warp to a group of sixteen cores, sixteen load/store units, or four SFU's. Because Warps execute independently, Fermi's scheduler does not need to check for dependencies from within the instruction stream. Using this elegant model of dual-issue, Fermi achieves near peak hardware performance.

Most instructions can be dual issued; two integer instructions, two floating instructions, or a mix of integer, floating point, load, store, and SFU instructions can be issued concurrently. Double precision instructions do not support dual dispatch with any other operation.

Figure 3.2 shows how the SP are distributed inside the SM and how the multiprocessors are distributed inside the GPU. Furthermore, Figure 3.3 shows the temporal evolution inside the SM and how it works for a block with 256 elements (`warp`$=256/32=8$ elements).

Any Thread can be labelled using `blockDim`, `blockId` and `threadId`. In an example with 14 Blocks and 256 Threads/Block (3584 elements), we find that for element 23 in Block 4, the labels inside the code are



(a) GF100 Streaming Multiprocessor (SM)

(b) 14-SM based Fermi Architecure detail

Figure 3.2: Description of our Fermi c2075 GPU based on GF100/GF110 Architecture.

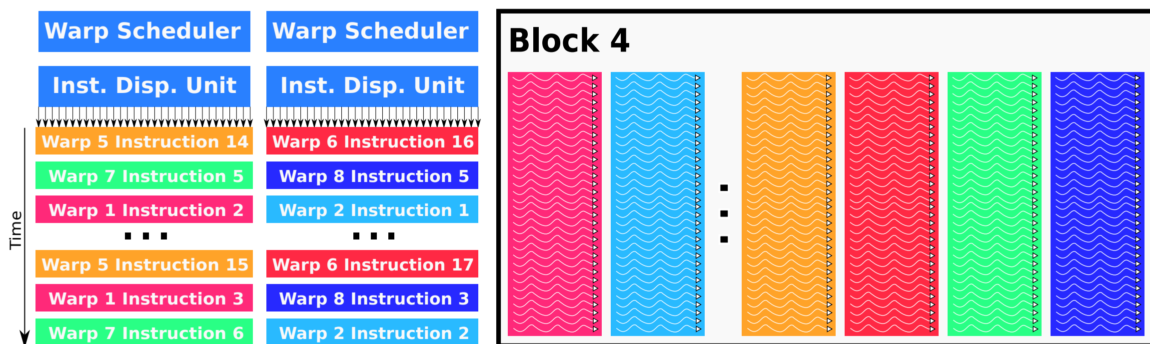


Figure 3.3: Execution pipeline for a Strem Multiprocessor (left) which process block number 4 (right)

- `blockDim=256`
- `blockId=4`
- `threadId=23`

and then, the typical access pattern, points to

```
i=threadId+blockDim*blockId=23+256*4=1047
```

3.3 CUDA development

The CUDA main functions are related to the memory interaction between CPU and GPU, in particular, `cudaMemcpy` with the different flags to stablish the way of the transfer. It is important to remark that these interactions or data transfers between GPU and CPU are extremely slow and should be minimized. Moreover, the allocation and memory freeing operations could be performed using their equivalences in CUDA as shown in listing 3.1

Listing 3.1: CUDA Most important functions

```
// GPU Memory allocation
cudaMalloc(...,size);
// GPU Memory free
cudaFree(..);
// Copy Host To device
cudaMemcpy(...,cudaMemcpyHostToDevice);
// Copy Device To Host
cudaMemcpy(...,cudaMemcpyDeviceToHost);
// Copy Device To device
cudaMemcpy(...,cudaMemcpyDeviceToDevice);
```

The advantage of using GPU for programming numerical methods, comes from the High-Level Single Instruction Multiple Data (SIMD) or as nVidia calls, Single Intructions Multiple Threads (SIMT) paradigm. Any operation can be executed in concurrence with many others allowing any CUDA Thread to access to a particular position while any other is accessing to another one.

3.3.1 Example of implementation in a 1D case

Consider, for example, the 1D transport equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{3.1}$$

with $c > 0$, and its initial and boundary conditions

$$u(x, 0) = f(x)$$

$$u(0, t) = U_0$$

applying the temporal discretization with forward Euler and the upwind scheme:

$$\frac{\Delta u_i}{\Delta t} = -\frac{u_i - u_{i-1}}{\delta x} \tag{3.2}$$

writing its as

$$u_i^{n+1} = u_i^n - \frac{u_i^n - u_{i-1}^n}{\delta x}\Delta t \cdot c \tag{3.3}$$

and the procedure could be written in Standard C as follows

Listing 3.2: Simple 1D transport equation in C

```
void upwindStepCPU(double *fn,double *fnmas1,double DELTAX){
  int i;
  for (i=1; i<1/DELTAX; i++) {
    fnmas1[i]=fn[i]+c*DELTAT*(fn[i-1]-fn[i])/(DELTAX);
  }
}
```

Since conventional processors are not-able to make this operation for any group of elements at the same time, the result will be obtained at the end of $1/\Delta x$ cycles. This kind of architecture is called SISD (Single Instruction Single Data) and it is used by the most common personal computers. The disadvantage of this implementation is the need of processing elements one-by-one, making easier the implementation of the code but not reaching good performance.

CUDA implementation of Listing 3.2 could be written as

Listing 3.3: Simple 1D transport equation in CUDA

```
1 __global__ void upwindStepGPU(double *fn,double *fnmas1,double DELTAX)
2 {
3    // Point to the data
4    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
5    if(i<MAX){
6     fnmas1[i]=fn[i]+c*(DELTAT*(fn[i-1]-fn[i])/(DELTAX);
7   }
8 }
```

The function invocation could be made as follows

Listing 3.4: Cuda functions calling

```
1   // CPU
2   f=(double*)malloc(sizeof(double)*MAX);
3   fnmas1=(double*)malloc(sizeof(double)*MAX);
4   ...
5   // Condiciones iniciales
6   ...
7   // Copia de CI a GPU
8   cudaMemcpy(d_f, f, sizeof(double)*MAX, cudaMemcpyHostToDevice );
9   cudaMemcpy(d_fnmas1, f_nmas1, sizeof(double)*MAX, cudaMemcpyHostToDevice );
10  ...
11  // Calculo
12  for (j=0; j<=(TFINAL/DELTAT); j++) {
13    upwindStepCPU(f,fnmas1,deltax);
14    reAsigna(f,fnmas1,deltax);
15  }
16  //GPU
17  for (j=0; j<=(TFINAL/DELTAT); j++) {
18    upwindStepGPU<<<blocks,threads>>>(d_f,d_fnmas1,d_deltax);
19    reAsignaG<<<blocks,threads>>>(d_f,d_fnmas1);
20  }
21  cudaMemcpy(f, d_f, sizeof(double)*MAX, cudaMemcpyDeviceToHost);
```

The key of this implementation is based on the fact that all Blocks and Threads together cover the amount of elements to be processed. The relation between Blocks (n_b), Threads (n_t)

and the amount of elements (n_e) must be

$$n_e \leq n_b * n_t \tag{3.4}$$

3.4 Results *All that glitters is not gold*

Recent work [7] [10] has been published reporting that it is possible to get Speed-Ups around 100x and 130x using Simple Precision Floating Point Data types. It is important to note that a few details must be taken into account before accepting this kind of results. Moreover, [16] explained that comparison tests from CPU to GPU, must be developed at the same conditions in order to be satisfactory. To take this into account the computational resources where tests are going to be performed in the present work are shown in Table 3.1 showing the computational facilities common to all the tests performed.

	CPU	GPU
Cores	6	448
Frequency (MHz)	2666.969	1150
DP R^{peak} (GFLOPS)	67.2	515.2
Memory (GB)	48	6
Mem. Bandwidth (GB/s)	32	144

Table 3.1: Intel Xeon X5650 @ 2.66 GHz and nVidia Tesla c2075 technical characteristics

With 1D transport equation, computational times as appear in table 3.2 can be obtained. The computational performance is function of the number of elements implied in the calculation, both for GPU or CPU implementations. This detail is very important when the number of operations increases and much more when the access to the main memory is high.

	1-Core		6-Core		GPU
n	t (ms)	Speed-Up	t (ms)	Speed-Up	t (ms)
1048576	143799.29	33.48	24844.87	5.78	4295.62
524288	71162.12	32.65	11931.02	5.47	2179.62
131072	17649.91	29.62	3034.35	5.09	597.98

Table 3.2: Computational performance through CPU (Mono-Core and Multi-Core) and GPU for t=(0,1), x=(0.0,1000.0), $\delta = 1000.0/n$ and $\Delta t = 10^{-4}$

The performance of the GPU is very high for the simplest 1D transport equation. The Speed-up has been measured as elapsed time at GPU divided by elapsed time at CPU. It reaches 33.48x for the mono-core version and 5.78x for the multi-core version. It implies a performance of around 73% with regard to the theoretical increase (7.66x and 46x).

It is widely accepted that Simple Precision has more throughput but it is not as precise as the Double Precision.

	GPU-DP			GPU-SP			GPU-SP2		
n	t (ms)	ϵ	S_{up}	t (ms)	ϵ	SS_{up}	t (ms)	ϵ	S_{up}
1048576	4293.87	-6.6437e-14	33.93	3345.54	-1.4267e-04	44.85	1904.97	-1.6645e-04	78.53
524288	2175.31	2.2146e-14	32.95	1716.82	4.7558e-05	42.91	981.00	1.1889e-05	75.59
131072	594.38	4.4291e-14	29.84	488.02	-1.0700e-04	37.61	293.64	-1.1889e-04	62.00

Table 3.3: Simulation time, accuracy and performance for GPU performing the calculations for using `Double`, `float` and a tuned version with `float` for t=(0,1), x=(0.0,1000.0), $\delta = 1000.0/n$ and $\Delta t = 10^{-4}$

In Table 3.3 shows three implementations, using `double` and `float` and a tuned version of `float` in order to exploit the benefits of the GPU when float is used. Hence, it is important to bear in mind that the use of the simple precision must be limited to those cases where the precision is not the most important aspect [12] but the performance is critical.

When quantifying the computational gain of GPU over CPU implementations, the following efficiency parameters are of interest:

$$\eta_{CPU} = \frac{R_{CPU}}{R_{CPU}^{peak}} \quad \eta_{GPU} = \frac{R_{GPU}}{R_{GPU}^{peak}} \tag{3.5}$$

where R and R^{peak} stand for the effective performance and peak performance of a particular configuration respectively. It is important to note that performance comparisons should be evaluated at similar individual levels of efficiency in both CPU and GPU implementations and, ideally, at maximum efficiency. However, it is not always easy to reach the ideal values of η_{CPU} =1 and η_{GPU} =1 of the processors, nor it is to ensure that both implementations offer $\eta_{CPU} = \eta_{GPU}$ prior to their comparison. On the other hand, it is worth noting that it is easier to improve the efficiency when working in GPU processors than in CPU implementations so that, frequently, comparisons are made between implementations where $\eta_{GPU} > \eta_{CPU}$. A good implementation in both architectures offers very similar results to the ones shown in the previous table.

The performance of the CUDA version could be obtained as follows

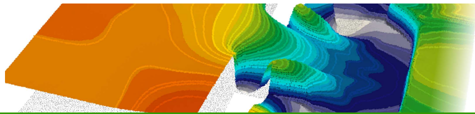
$$\gamma = \frac{S_{up}}{S_{up}^{Theoretical}} = \frac{t_{GPU} R_{CPU}^{peak}}{t_{CPU} R_{GPU}^{peak}} \tag{3.6}$$

Attending to this implementation, it is obtained a relation of 73% of efficiency in the implementation of the GPU using `Double` data type and 85% using the tuned `float` version. When reading some literature, 140x is affordable [7] but we suggest that it is very important to analyze the results and to apply some common sense.

[7] obtains gainances about 21x using Double Precision data types. It is used a Intel Xeon E5430 (2.66 GHz, 12 MB L2 Cache) which achieves R_{CPU}^{peak} = 10.689 GFLOPS/core (4 Cores) and a nVidia GeForce GTX 260, which has $R_{CPU}^{peak} \approx 71$ GFLOPS. For this configuration, the ratio of the theoretical maximum gainance, assuming $\eta_{GPU} = \eta_{CPU} = 1$,

$$\frac{R_{GPU}^{peak}}{R_{CPU}^{peak}} = 6.64 \tag{3.7}$$

In the work, 21 gainance has been shown, which implies $\gamma > 3$, implying that $\eta_{GPU} >> \eta_{CPU}$. When both CPU and GPU implementations are mostly optimized, this performance is overstimated and we propose that $\gamma \approx 1$ is a very acceptable performance, showing the profits of the GPU and not taking to confusion to developers. In this work, we have tuned both implementations in order to show a realistic performance of the GPU implementation.



4 Implementation

The implementation and its difficulty is not the main topic of this work but some interesting details are explained that could be useful in any other application of a similar explicit finite-volume scheme. In particular, details about the importance of and how to obtain memory coalescing profits, solving bottle-neck problems or writing output files with the minimum penalty are described below. They are all related with the necessity to avoid data transfer between the GPU and the CPU during the calculation as much as possible.

4.1 Model overview

The main of the implementation is shown in Listing 4.1. There it is shown the main aspects of the programming and the general aspect of any similar code.

Listing 4.1: Overview of the CUDA implementation.

```
...
// Configuration of the parameters
threads=512;
wallBlocks=nWall/threads;
cellBlocks=nCell/threads;
while(t<tmax){
  // Calculate the fluxes
  calculateWallFluxes<<<wallBlocks,threads,0,executionStream>>>(...);
  // Stablish the minimum dt obtaining the ID of the
  // minimum dt
  // (*) Explained at section 4.3
  cublasIdamin(...,nWall,vDt,1,id);
  // And assign it
  newDt<<<1,1,0,executionStream>>>(dt,vDt,id);
  // Update the elapsed time (in GPU)
  updateT<<<1,1,0,executionStream>>>(cuda_t,dt);
  // Retrieves the value of t to CPU
  cudaMemcpy(t,cuda_t,sizeof(double),cudaMemcpyDeviceToHost);
```

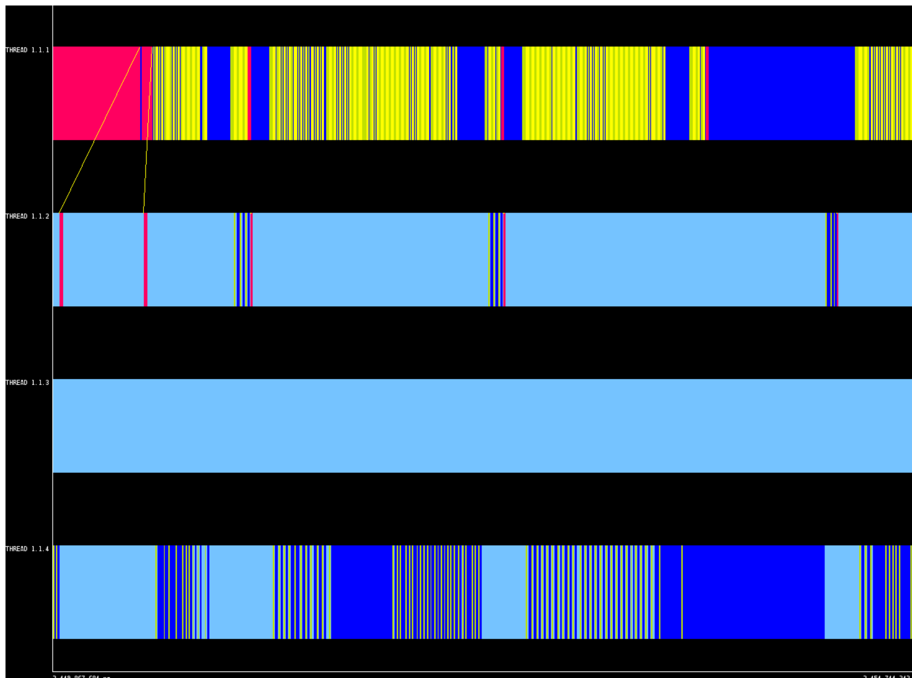


Figure 4.1: Execution trace and performance detail for a time-step using Paraver

```
19    // Update the cell values
20    assignFluxes<<<cellBlocks,threads,0,executionStream>>>(...);
21    // Verify if it is neccessary to dump data and
22    // if it is neccessary, process it.
23    // (*) Detailed in section 4.4
24    if(t<t_dump){
25      // Copy of cell variables to a GPU
26      // stored buffer
27      cudaMemcpy(..., cudaMemcpyDeviceToDevice);
28      // Stablishing the barrier to ensure the copy of the
29      // data to the buffer
30      cudaStreamSynchronize(copyStream);
31      // Copy the data to the CPU buffer
32      cudaMemcpyAsync(..., cudaMemcpyDeviceToHost,copyStream);
33      // Create another stream in order to be which controls
34      // the disk-transfer
35      pthread_create( &diskThread, ...);
36    }
37  }
```

The details of this implementation are described below. Furthermore, the behaviour of the code is described in a timeline which trace has been obtained using Paraver (*www.bsc.es/computer-sciences/performance-tools/paraver*) in Figure 4.1.

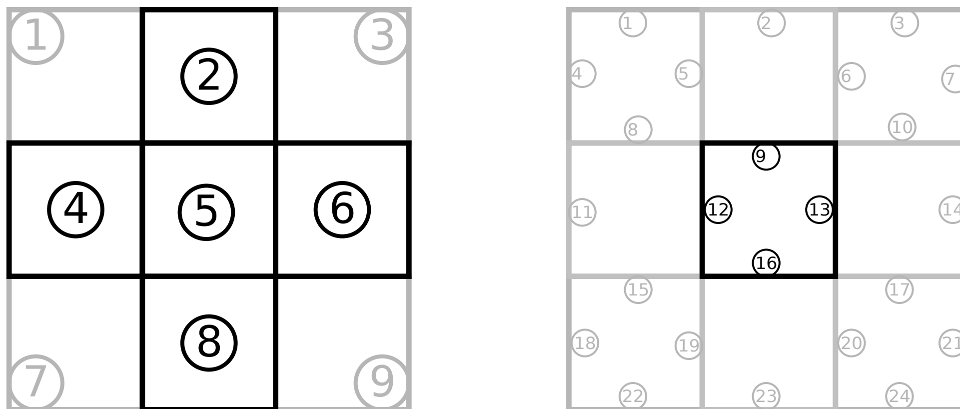


Figure 4.2: Structured mesh with Cell Numbering detail (Right) and Wall Numbering detail (Left) example

4.2 Memory coalescing

Memory coalescing is the way the memory is ordered allowing half-Warp to access global memory at the same time (using only 1 cycle to perform the load operation). This means that, if a Thread (The first one) in a Warp accesses to a particular memory address and it access pattern is such that access to the next address (`i, i+1, i+2....`) the following 31 Threads do not need to read the memory again. Otherwise, two or more accesses are needed to allow each Thread the access to data.

Memory coalescing is one of the most important things to take into account when programming GPU's. Recent works [29] have demonstrated the efficiency of coalescing techniques, being this implementation better in some cases than shared memory strategies. Although there exist works dealing with the profits of using this strategy, the way to proceed when using unstructured meshes is not clear. This topic will be discussed in the next May 2012 *GPU Technology Conference* [8] and some improvements are detailed in [25].

In our case, the perfect memory coalescing technique could be implemented, [5] [7], if using structured meshes. As it appears in Figure 4.2, cell labelling implies that the access pattern for a Block of (in this case 9) cells allows the programmer to make the perfect match access into a Warp. In other words, for any group of cells within a Warp, all the variables are accessible in only a coalesced reading.

Being the present work oriented to a general implementation of the finite volume scheme on both structured and unstructured grids, the memory optimization is not as easy as described above.

According to the general updating formula 2.30, this scheme works with the cell edge fluxes or inter-cell elements through which the Rienmann Problem is solved. In the case of the structured mesh, this flux takes place into the left, right, upside and downside cell to a given cell, so all the operations could be performed looping by cells. In unstructured grids, this concept is different

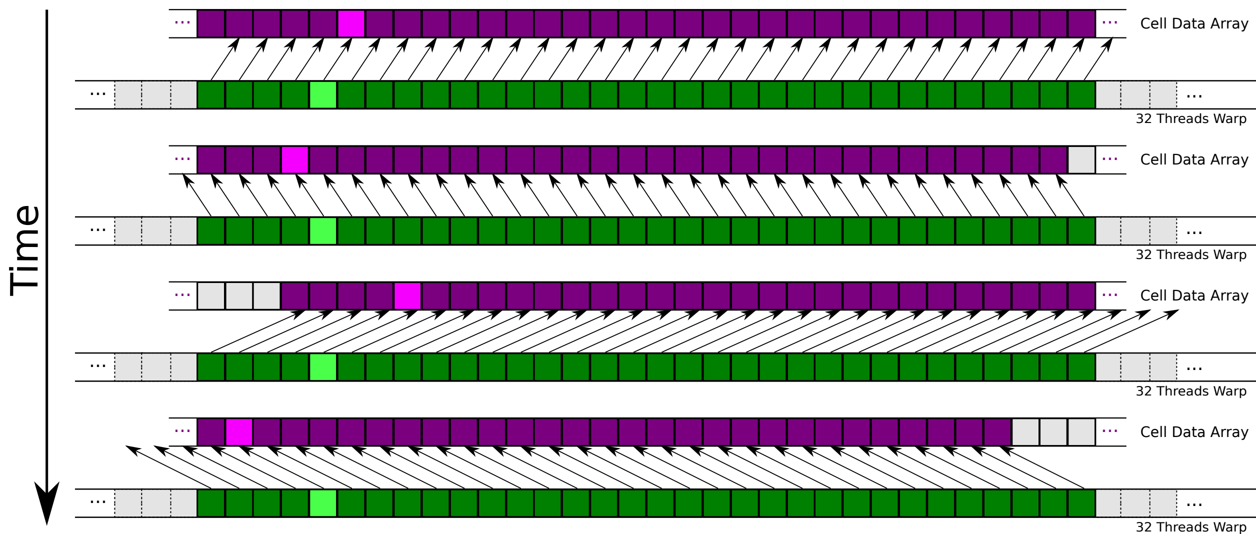


Figure 4.3: Misaligned and Coalesced access pattern to compute the flux variation for any group of elements following the scheme of Right, Left, Down, Up for W data (Stored by cell) in a mesh ordered as Figure 4.2. Light coloured correspond to the processed element 5, wich implies cells 2, 4, 6 and 8.

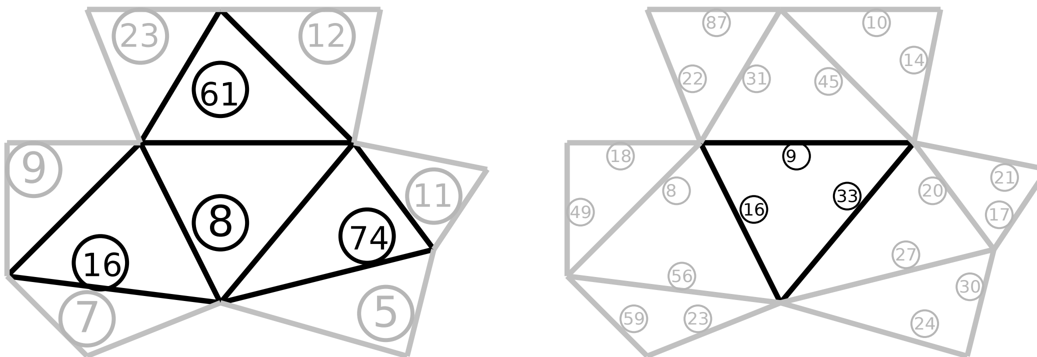


Figure 4.4: Unstructured mesh with Cell Numbering detail (Right) and Wall Numbering detail (Left) example

and it is good idea to make the flux calculations by walls and then, to assign them to each cell with the need to keep trace via a connectivity matrix.

For the general unstructured case it is important to decide how to stablish the order of the variables. It can be performed through cells or through walls. Using as example Figure 4.2, the operations of applying the variation to the cell (8) has no a coalesced pattern. There exists the need of searching the neighbouring cells (74,61,16) and calculating the flux through walls (33,9,16). Sketching these operations in an example for wall 33 (i=33, c1=8, c2=74) we have:

Listing 4.2: Access pattern for the main flux variation operation.

```
calculateWallFluxes(...){
// Loop by wall
int i = threadIdx.x+(blockIdx.x*blockDim.x);
if(i<nWall){
```

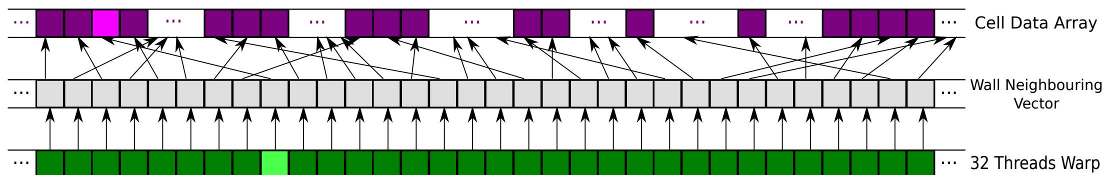


Figure 4.5: Uncoalesced access pattern to get W data (Stored by cell). Processing wall 9 is light coloured when it accesses to cell 8 (i=9, c1=8)

```
5       c1=wall[i];
6       c2=wall[i+1];
7       // [COALESCED] Access to the variables of the wall
8       // Normal Vector
9       // Length of the side
10      // ...
11      ...
12      // [UNCOALESCED] Access to the variables of c1 and c2
13      // Primitives variables
14      // Area of the cells
15      // ...
16      ...
17      // Store the value of the flux for the wall i
18    }
19  }
```

Although this is the main function where the flux is calculated and it involves many uncoalesced accesses to the variables, there are some operations whose access could be performed through the cells.

4.3 Gathering data avoiding bottleneck

One of the troubles when trying to make all the operations inside the GPU is the identification of global quantities such as the minimum value of a vector. As the Many-Core paradigm is not designed to share information between elements, reduction operations like `min, max, sum...` are performed at `cublas` library [23].

`cublas` library has high-level functions that work retrieving results to GPU or to CPU. When interested in using them without taking out the data from the GPU, that must be specified. This could be done through `cublasSetPointerMode_v2(handle, CUBLAS_POINTER_MODE_DEVICE)`, stating that all results have to be returned to the GPU memory.

In our case, it is essential that the algorithm calculates the minimum Δt following the CFL condition when running along all the cell edges. Then, following Figure 4.3 scheme, the minimum among all of them is selected. Details are shown in Listing 4.3.

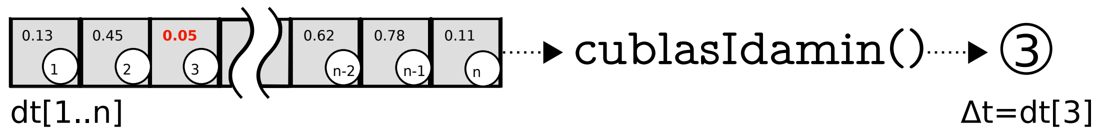


Figure 4.6: Gathering minimum Δt for all the domain

Listing 4.3: Gathering Δt operation

```
1 __global__ void newDt(double *dt,double *vDt, int *id){
2    // As cublasIdamin returns it value following
3    // 1-based indexing, we must to substrate 1
4    *dt=vDt[*id-1];
5 }
6  ....
7  cublasIdamin(handle,*npared,vDt,1,id);
8  newDt<<<1,1>>>(dt,vDt,d_id);
9  ...
```

While calculation is controlled by host, it is necessary to transfer the updated t^{n+1}. After δt is calculated, the updating operation can be perform as 4.4 and then, you can transfer the updated value of t^{n+1} to CPU.

Listing 4.4: Updating Δt

```
1 __global__ void updateT(double *dt,double *t){
2    int i;
3    *t=*t+*dt;
4 }
```

In order to calculate the global mass error, there is a sum of mass inside the mesh and the balance between the inlet and outlet boundaries

$$\mathbf{M} = \rho \sum h_i A_i \tag{4.1}$$

and then, it calculates the error as

$$\epsilon = \frac{\mathbf{M}^{n+1} - \mathbf{M}^n + \mathbf{M}_{in} - \mathbf{M}_{out}}{\mathbf{M}^{n+1}} \tag{4.2}$$

The sums are performed using `cublasDasum` where all elements are added within a vector and the results stored in a variable, working similar to `cublasIdamin`.

4.4 Writing output files

The feature of the newest CUDA models allowing for simultaneous execution and copy streams can be used to hide delays caused by writing data to disk.

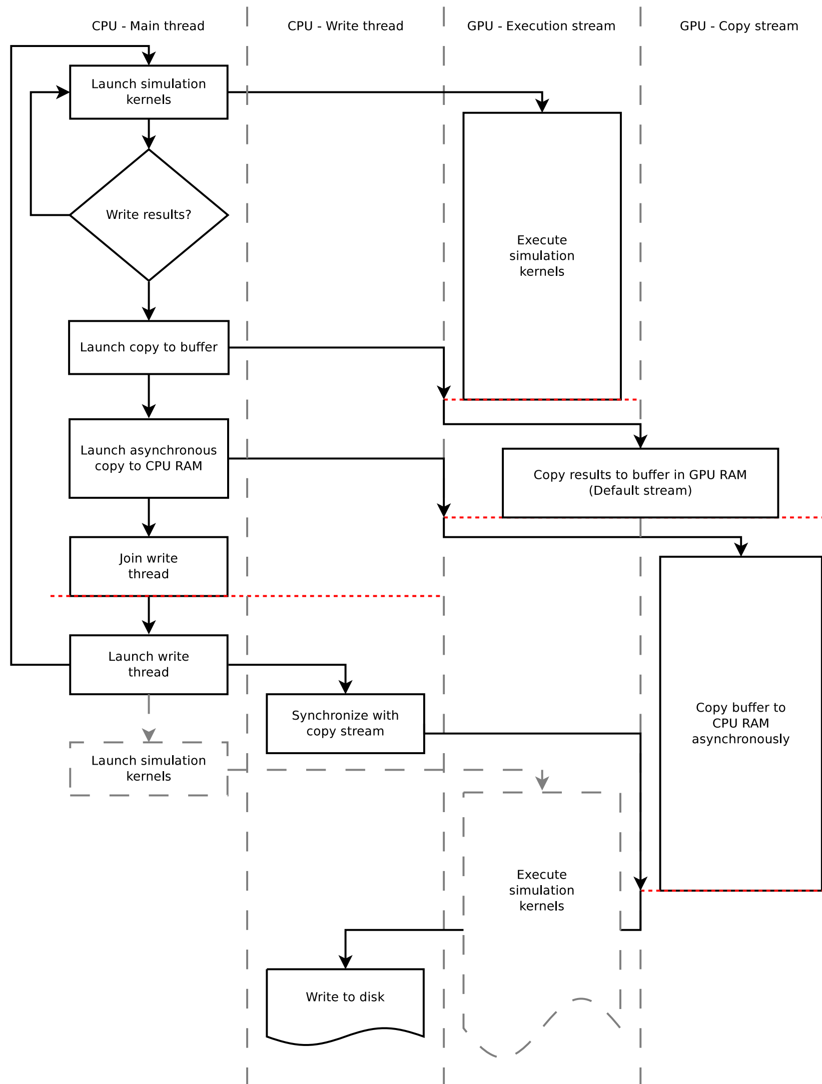


Figure 4.7: Asynchronus dumping data diagram.

Traditional cudaMemcpy performs a synchronous copy, i.e., the call does not return until the copy is complete. However, calls to the new family of asynchronous functions like cudaMemcpyAsync may return before the copy is complete. Furthermore, the copy may be assigned to a stream. In this way it is possible for the CPU host code to call cudaMemcpyAsync and assign it to a copy stream, then launch kernels in an execution stream. Both streams are processed simultaneously by the GPU.

It is not possible to use cudaMemcpyAsync directly to copy simulation results to Host memory in the case of shallow flow simulation because the concurrent simulation would alter the values in the variables being copied. It is necessary to make a synchronous copy to a buffer in GPU memory first (Figure 4.7). Once the copy of the results to the buffer is complete, a call to cudaMemcpyAsync is made which copies the buffer to host memory, and the simulation kernels are launched simultaneously operating on the usual variables.

This scheme requires that the CPU launches kernels after the call to the asynchronous copy. It is necessary to introduce a parallel CPU thread that waits for the copy to finish and then writes the results to disk. Thus, the main CPU thread will first call cudaMemcpyAsync, then spawn a write thread and continue launching kernels to advance the simulation. The first task for the writing CPU thread will be to wait for the copy stream to finish, then proceed to write the results in host memory to disk.

The limitation in this scheme is that the computation time between dumps to disk has to be greater than the writing time to disk itself. If that is not the case, gains can still be achieved from using this scheme but further barriers are required. One of them is that the main CPU thread has to wait for the writing thread to finish before calling cudaMemcpyAsync. Depending on the problem, further gains can be made e.g. using multibuffering.

4.5 Compilation and other issues

In the original Fortran version of the code there are several functions related to the preprocess and postprocess as sketched on figure 4.8. To be more efficient, the programming of that part of the code in C has been ommitted and the work has focused on the efficient programming of the numerical aspects. So the preprocess is performed through the Fortran version and the computing kernel is performed using C/CUDA.

To work with the two codes at the same time, they have been compiled together. The technique used is based on making a standard C interface which interoperates with CUDA and is called from Fortran as shown in [1]. The most complicated and interesting detail of this operation is the way of compiling them. It is shown in Listing 4.5.

Listing 4.5: Makefile Script

```
1
2  NVCC = nvcc
3  FORT = gfortran
4
5  FORTFLAGS = -w -O3
6  CUFLAGS = -g -w -O3 -m64 -arch sm_21 -Xptxas -dlcm=ca -I$(EXTRAE_HOME)/include
7  LDFLAGS = -L/opt/cuda/4.0/lib64 -L$(EXTRAE_HOME)/lib -lcudatrace -lcuda -lcudart
       -lstdc++ -lcublas -lrt -lm -lpthread
8  OBJ = cuda_blocks2mf.o SFS2Dv01_64.o
9  BIN = sfsGPU
10
11 $(BIN): $(OBJ)
12   $(FORT) $(FORTFLAGS) $(OBJ) $(LDFLAGS) -o $@
13
14 clean:
15   $(RM) $(OBJ)
```

```
16
17 cleanEx:
18   $(RM) $(OBJ) $(BIN)
19
20 cuda_actualiza.o: cuda_actualiza.cu
21   $(NVCC) $(CUFLAGS) $< -c -o $@
22
23 cuda_blocks2mf.o: cuda_blocks2mf.cu
24   $(NVCC) $(CUFLAGS) $< -c -o $@
25
26 SFS2Dv01_64.o: SFS2Dv01_64.for
27   $(FORT) $(FORTFLAGS) $< -c -o $@
```

Bearing in mind that all the structures are created as Vectors in Fortran and Fortran indexing are 1-based (C uses 0-Based) an special access is required (Eq (4.5), (4.5) and (4.5)). Furthermore, Fortran stores the elements following Column-Major Order while C storing is Row-Major Order based. These two aspects imply that:

- The access to the particular position i of array $V[M]$ is made, in C, as

$$V(i) = V[i-1] \tag{4.3}$$

- The access to the particular position i, j of array $V[MxN]$ is made in C as

$$V(i,j) = V[(j-1)\cdot M + i - 1] \tag{4.4}$$

- The access to the particular position i, j, k of array $V[MxNxO]$ is made in C as

$$V(i,j,k) = V[(k-1)\cdot M \cdot N + (j-1)M + i - 1] \tag{4.5}$$

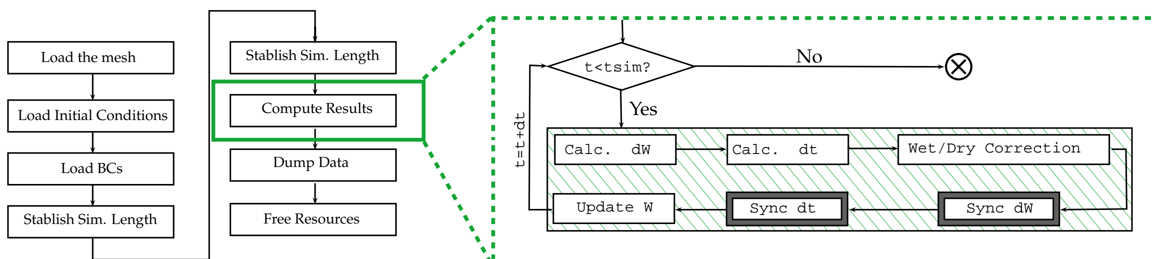
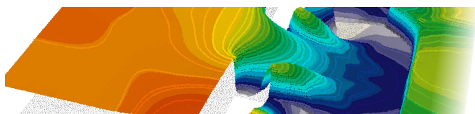


Figure 4.8: Flux diagram for the application. Green-highlighted is the ported slice of the code



5

Results

The cases chosen to show the results are focused on how similar are the GPU numerical results to the ones obtained from the original CPU version (precision) and how efficient this implementation can be (performance). To achieve this, two examples have beed selected. First, an academic case of unsteady flow with source terms with analytical solution and second a real life inundation flow of hydraulic interest. Furthermore, the GPU perfomance has been compared with that of a distributed-parallel version of the CPU code at [14] using a dam-break flow simulation with a large number of cells.

5.1 Precision: A test-case with analytical solution

This case has been used to minimize the differences between the results provided by the CPU and the GPU versions. The case simulates the evolution of a mass of water contained in a frictionless paraboloid. Test Case 1 corresponds to zero initial velocity and a curved initial free surface shape (Figure 5.1). As times goes on, the potential energy transforms into kinetic energy. It is a good case because it has analytical solution [27] and there exists a challenging wet/dry boundary all the time.

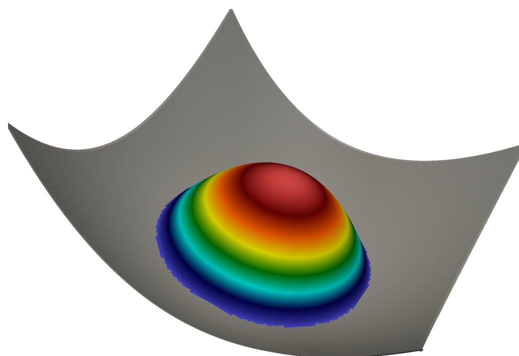


Figure 5.1: Left: Bed level and initial water depth state for test case 1.

As shown in Figure 5.2 and Figure 5.3 there are not visible differences between both simulations. In order to quantify the precision of the GPU implementation with respect to the CPU,

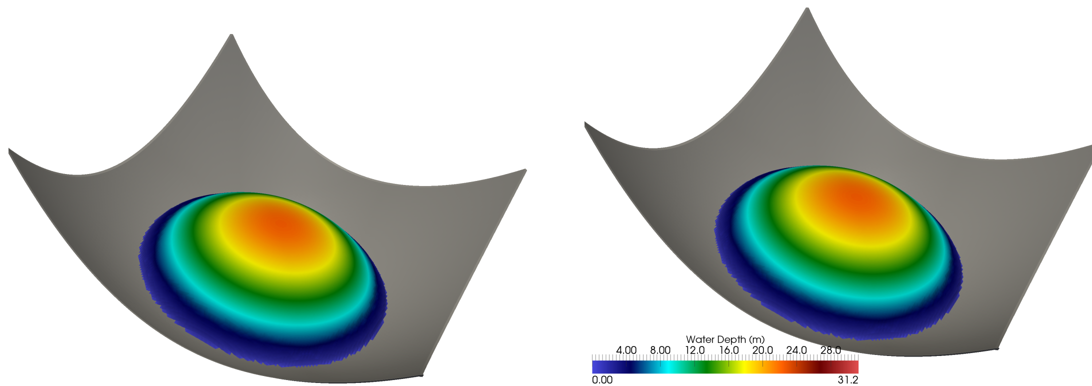


Figure 5.2: Test case 1. Left: GPU Simulated results for h and Right: CPU Simulated results for h at $t = 42.03s$.

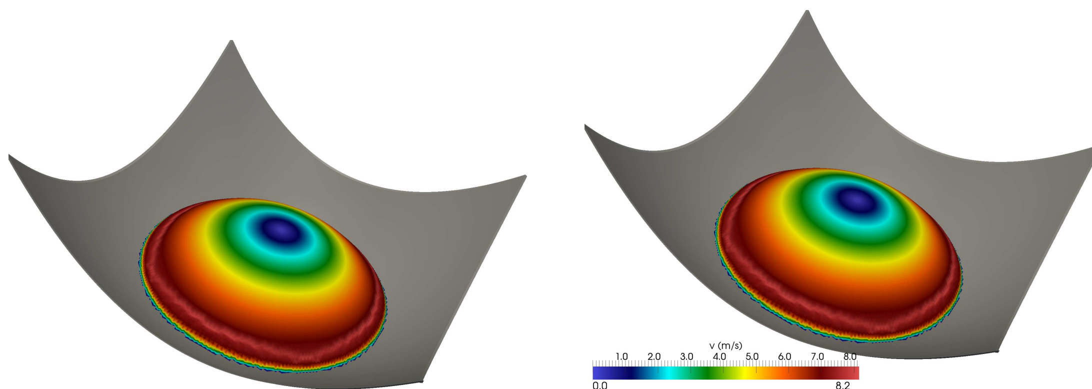


Figure 5.3: Test case 1. Left: GPU Simulated result for $|v|$ and Right: CPU Simulated results for $|v|$ at $t = 42.03s$.

the L_1, L_2 and L_∞ norm of the error in water depth at different times has been calculated. Test Case 1 shows acceptable differences. This agrees with the error in the calculation reaching machine precission ($\mathcal{O}(-14)$) in both versions of the code. The most sensitive region is the wet/dry boundary where both the water depth and velocity are very small.

Test Case 2 corresponds to the same frictionless container but with different initial data corresponding to a flat surface with velocity. Although the visual comparison is also favorable, the detailed evaluation of the L_1, L_2 and L_∞ norm of the error in water depth at different times shows unacceptable differences which come from the precision of the `double` floating point data type, reaching $\mathcal{O}(L_\infty) = -4$. Studying the procedence of the differences we find the problem at the first time step (See Figure 5.1).

Following the numerical scheme, we found that:

$$h_j^{***} = h_j^n - \alpha_k^1 + \left(\frac{\beta}{\tilde{\lambda}}\right)_k^1 \geq 0 \tag{5.1}$$

Attending to the new state for the second time-step, we found the values for cell `65399` as appears in Table 5.2

		L_1 Norm	L_2 Norm	L_∞ Norm
Test Case 1	T/4	5.8354e-06	4.4112e-08	5.0000e-10
	T/2	8.0286e-06	5.1624e-08	4.9991e-10
	3T/4	8.0451e-06	5.1698e-08	4.9995e-10
	T	7.9398e-06	5.1307e-08	4.9988e-10
Test Case 2	T/4	1.4805e+01	2.9783e-01	5.9207e-02
	T/2	1.6664e+01	2.2877e-01	3.1504e-02
	3T/4	1.7416e+01	3.3210e-01	1.1387e-01
	T	2.5452e+01	4.3794e-01	5.4876e-02

Table 5.1: L_1, L_2 and L_∞ for h

	CPU	GPU
α	-7.00000000000000188e-03	-7.0000000000000045e-03
β	-1.83403406456914518e-03	-1.8340340645691430e-03
λ	2.62004866367020641e-01	2.6200486636702058e-01
$\epsilon \propto -\alpha_k^1 + (\beta/\tilde{\lambda})_k^1$	-8.67361737988403547e-19	8.6736173798840355e-18
h_i^{n+1}	9.7990000000000005e-06	6.7151999999999997e-05
$L_\infty(h_i)$	5.7353e-05	

Table 5.2: Computational results in the first time step for α, β, λ, h and L_∞ for h in the conflictive cell

Although there are little differences, visual results appear to be the same as it is shown in figures 5.5, Figure 5.6 and Figure 5.7

In order to avoid this differences in computational accuracy and the corresponding non-physical fluxes, the following restriction is included (where hls=h^* and hrs=h^{***}).

Listing 5.1: Access pattern for the main flux variation operation.

```
1 ...
2   if(hls<COTAMIN1_15)
3     hls=0.0;
4   if(hrs<COTAMIN1_15)
5     hrs=0.0;
```

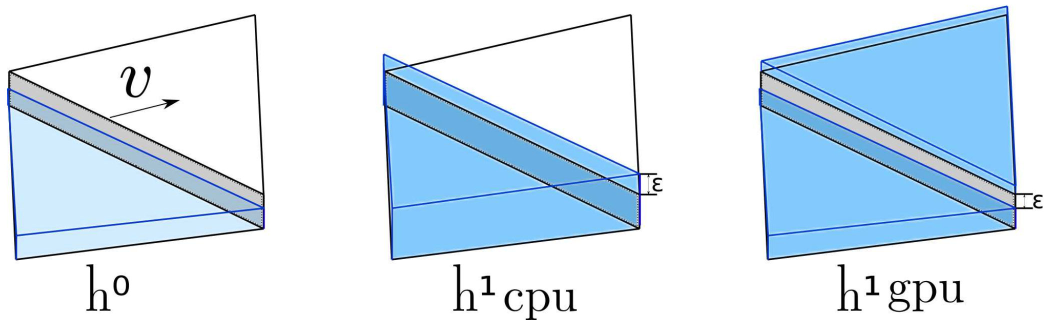


Figure 5.4: Left: Initial state h_0 for the conflictive cell. Center: h^1 for CPU. ϵ accuracy involves wall treatment as solid edge implying an increasing in it water depth. Right: h^1 for GPU. ϵ accuracy involves wall treatment as non solid edge so that water level decrease at cell i and increase at cell j.

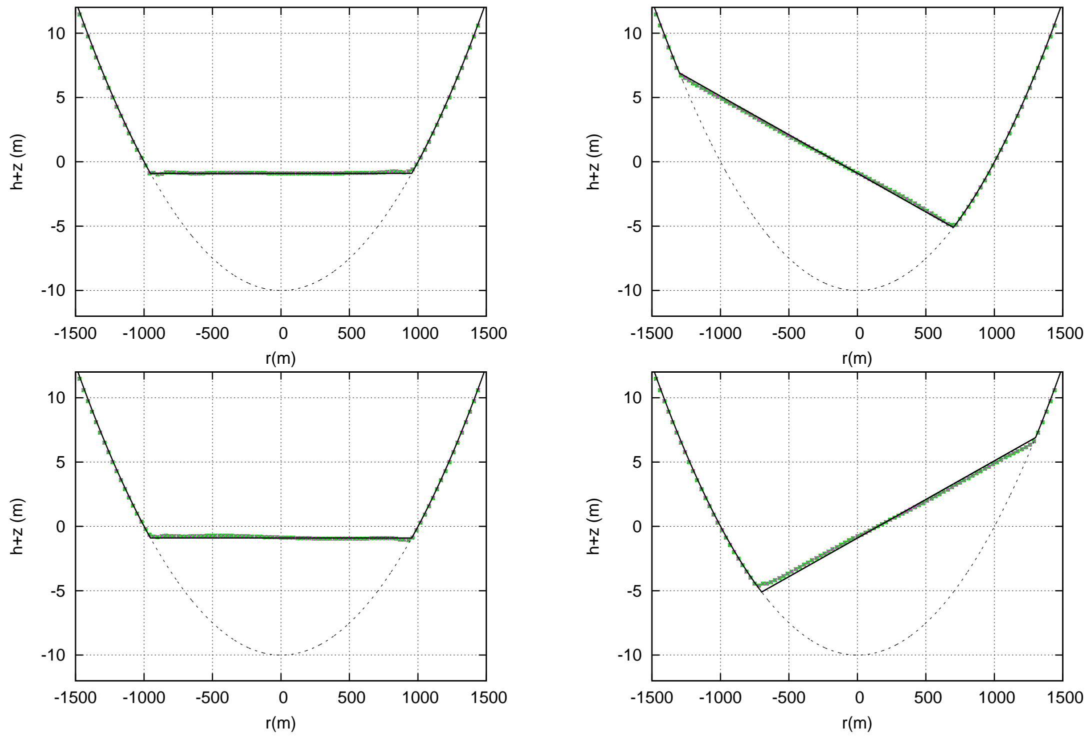


Figure 5.5: From Left to right, Top to down, h+z for t=T/4, T/2, 3T/4 and T

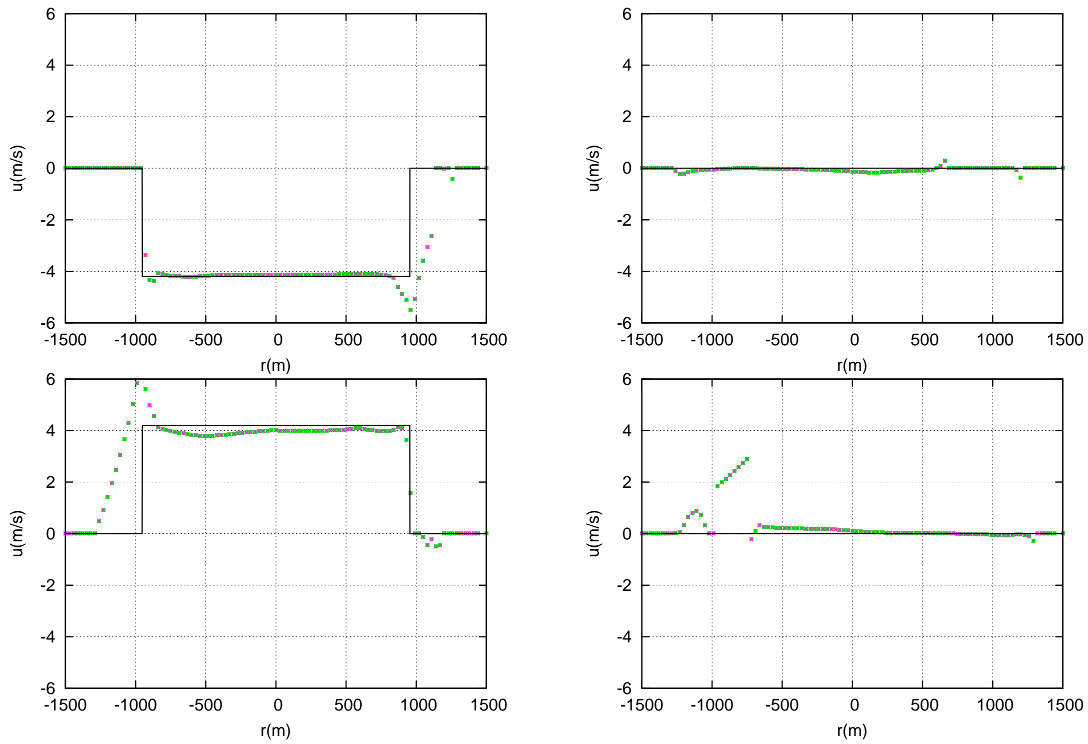


Figure 5.6: From Left to right, Top to down, u for t=T/4, T/2, 3T/4 and T

[6] . . .

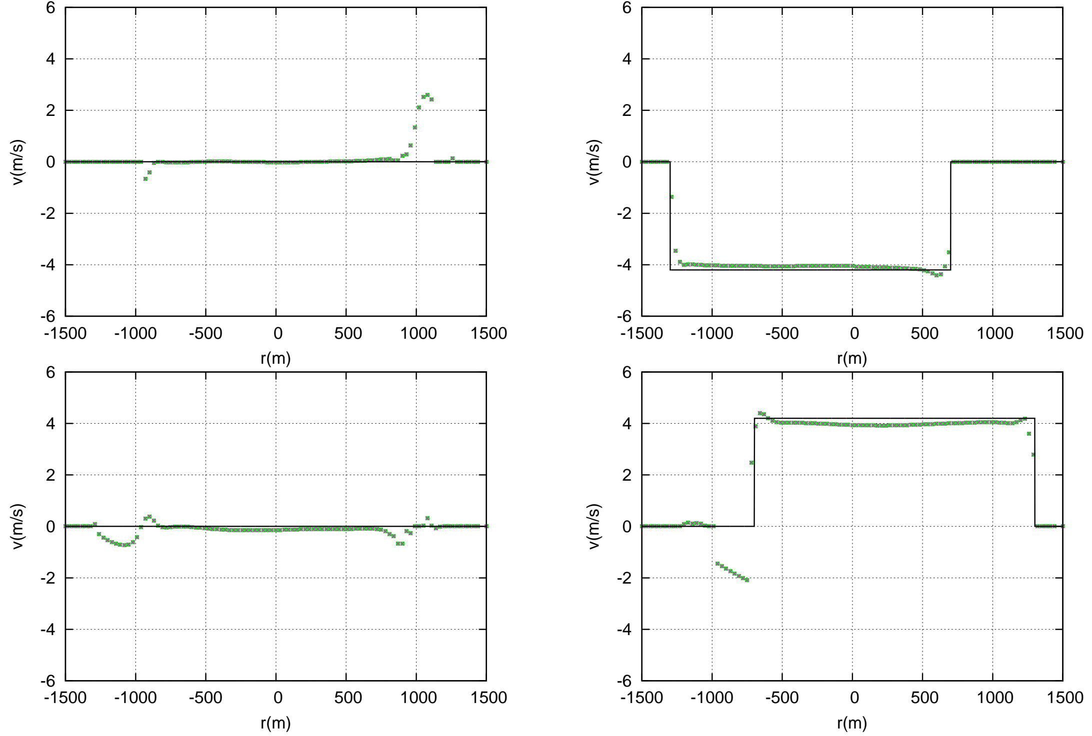


Figure 5.7: From Left to right, Top to down, v for t=T/4, T/2, 3T/4 and T

With this correction, the following norms are obtained (See Table 5.3)

		L_1 Norm	L_2 Norm	L_∞ Norm
Test Case 1	T/4	3.1995e-11	5.6889e-13	2.1316e-14
	T/2	3.4019e-11	5.8436e-13	1.0658e-14
	3T/4	4.4666e-11	6.7184e-13	1.7764e-14
	T	3.3531e-11	5.8323e-13	1.9984e-14
Test Case 2	T/4	2.0493e-11	4.5302e-13	1.0658e-14
	T/2	3.4429e-11	5.8694e-13	1.0658e-14
	3T/4	3.2590e-11	5.7146e-13	1.0658e-14
	T	3.3222e-11	5.7687e-13	1.0658e-14

Table 5.3: L_1, L_2 and L_∞ for h before applied the correction

5.2 Performance: A large-scale simulation at Júcar River

A realistic case with a long simulation time has been used in order to study the behaviour of the implementation in a large spatial and time scale case. Tous Dam is the last flood control structure of the Júcar River basin in the central part of the Mediterranean coast of Spain. During the 20th and the 21st October 1982 a particular meteorological condition led to extremely heavy rainfall. As a result the Júcar River basin suffered flooding all along and the Tous Dam failed with devastating effects downstream. The first affected town was Sumacárcel, about 5 km downstream of Tous Dam, lying at the toe of a hill on the right bank of Júcar river [3].

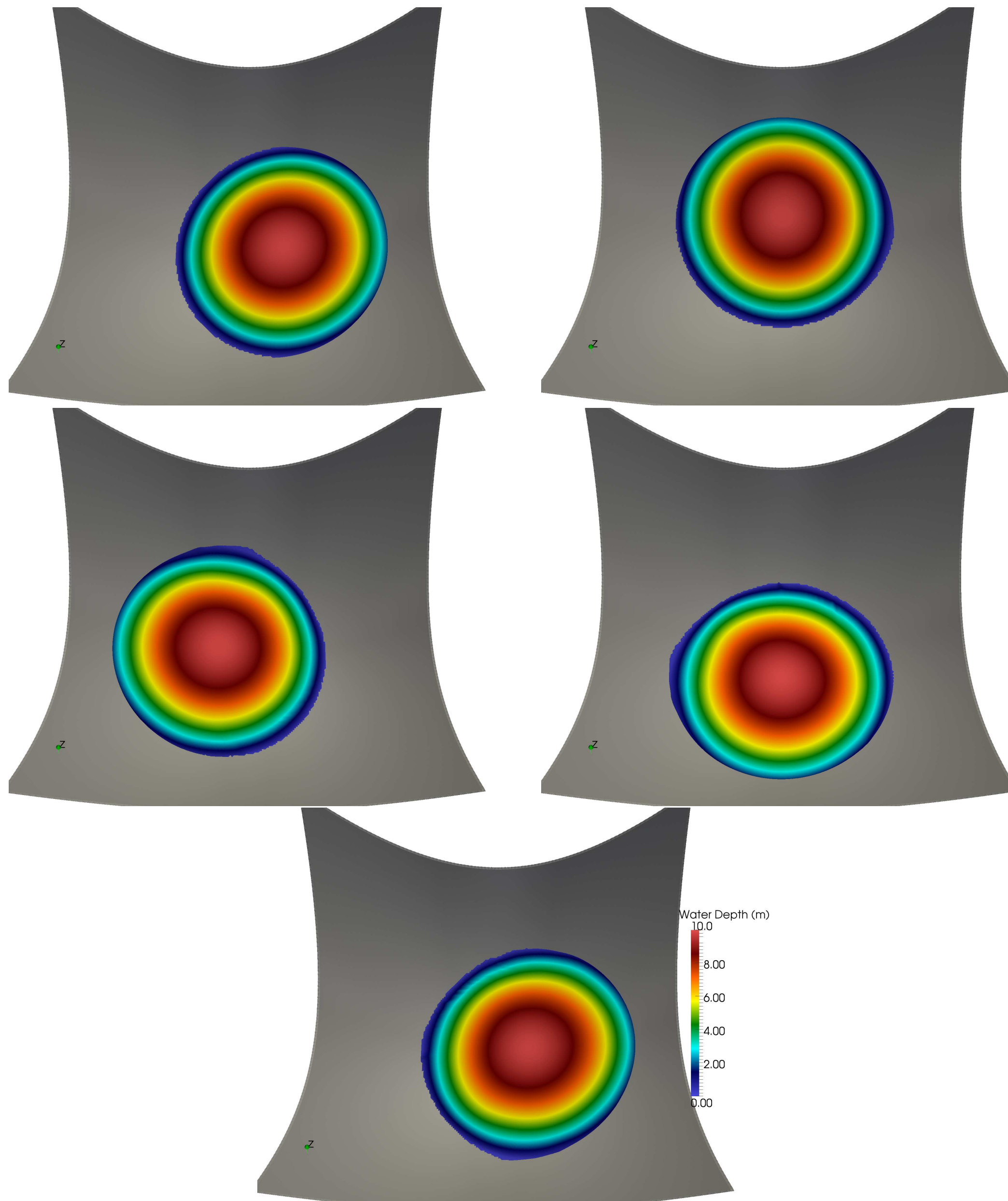


Figure 5.8: From Left to right, Top to down, h for t=0, T/4, T/2, 3T/4 and T

The terrain is moderately mountainous and most of the buildings lie on a slope that partially protected them from the flood. The ancient part of the village, however, is located closer to the river course and was completely flooded, with high water marks reaching between 6 m and 7 m.

The resolution of the available topographic data allow flood modelling. The DTM model used in this work was generated by CEDEX in 1998 [3]. From this information two numerical domains of different size and grid refinement were defined. The first domain, wich we will refer to as D_1, covers most of the original DTM, starting just after the dam location and finishing approximately 1 km downstream of Sumacárcel. More details can be found in [3].

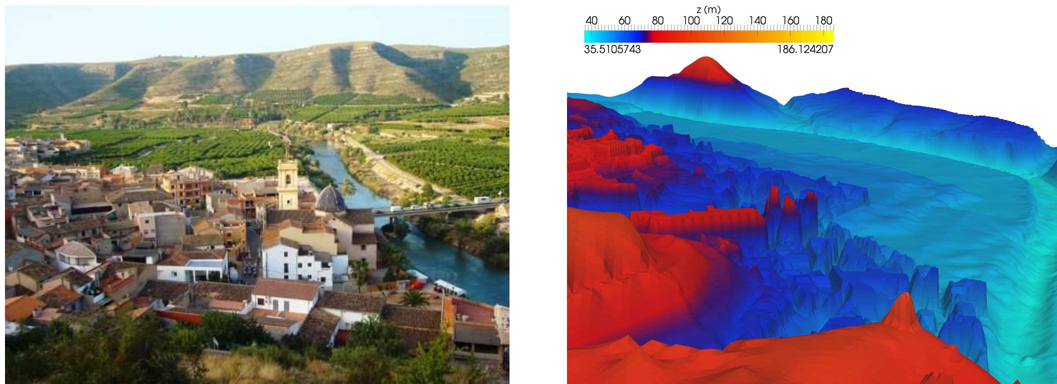


Figure 5.9: Left: Sumácarcel photography. Right: simulation mesh

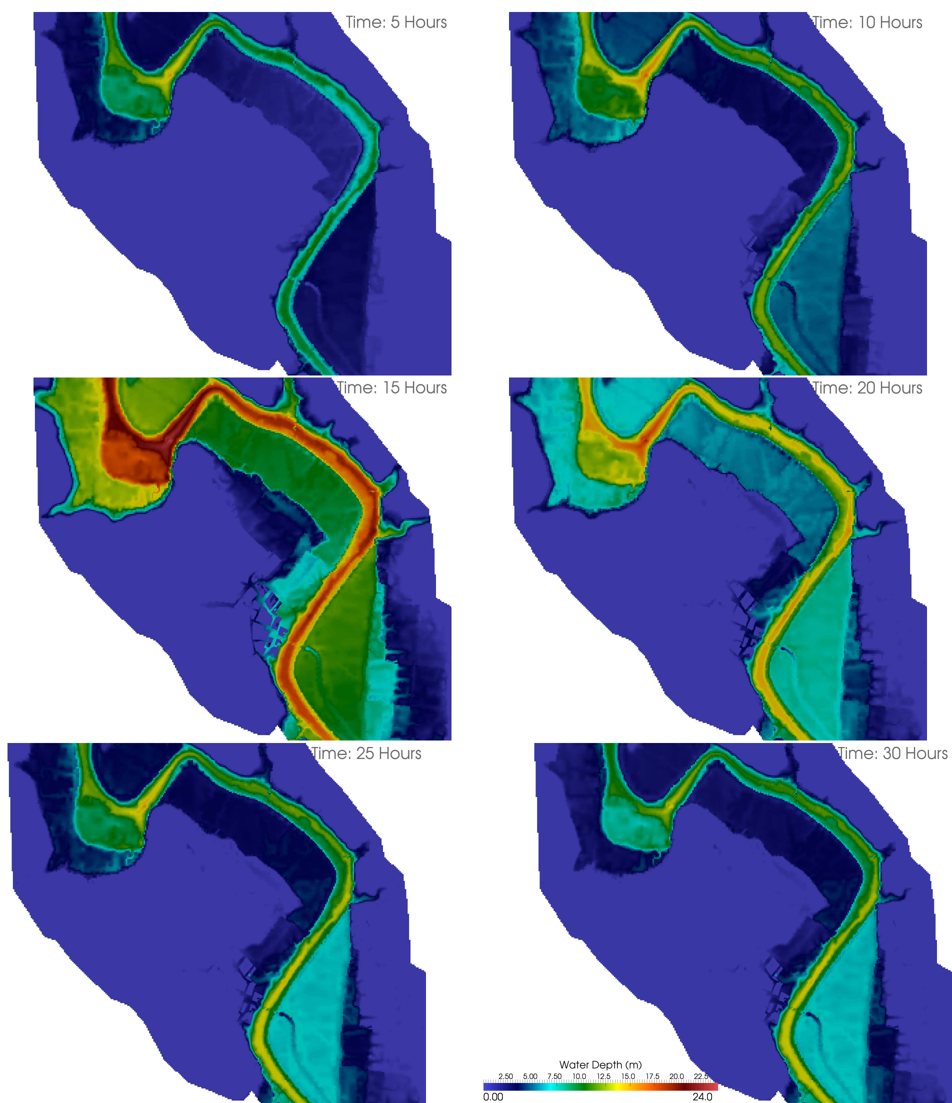


Figure 5.10: Water depth evolution for (Left-right, Top-down) $t = 5, 10, 15, 20, 25, 30h$

	CPU		GPU
Cells		563712	
CFL		0.9	
t_n		140400.0	
Q_0		$0\ m^3/s$	
Comp. Load (h.)	698.52		22.31
S_{up}		**31.31**	

Table 5.4: Simulation time for test case 2

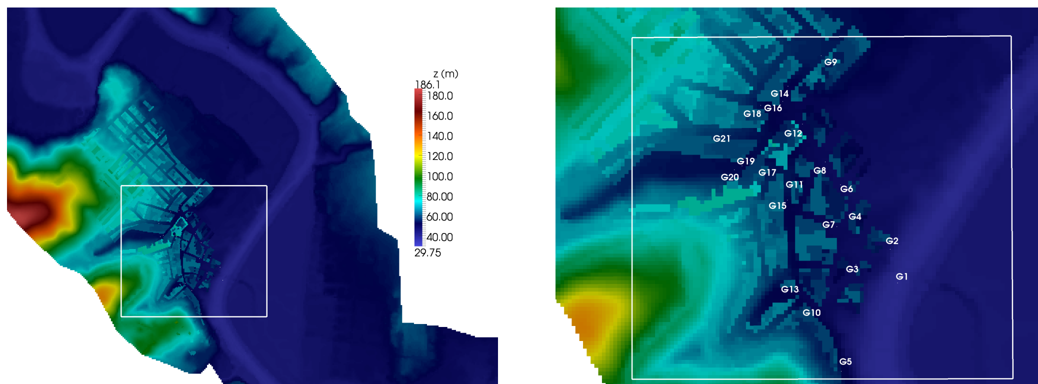


Figure 5.11: Gauges position

The values of reference to evaluate the quality of the simulations are field data of the maximum level reached by the flood wave at different locations within the town [3]. The location of these gauging points is shown in Figure 5.11.

D_1 was constructed using a triangular structured mesh with side length 5 m, able to provide a correct representation of the village. This led to 144669 grid cells. When doubling the cell size the resolution of the buildings was smeared and the village topography was poorly defined, providing an unrealistic definition of the problem.

The second discretization D_2, covers a small part of D_1, focusing on the representation of the village and was generated using a finer structured triangular mesh characterized by cell sides of 2.5 m over a smaller domain (grid density increased by a factor 4). This discretization involves 563712 cells. Both discretizations D_1 and D_2 are able to reproduce the narrow streets of the village, although the mesh D_2 provides a sharper delimitation of the buildings.

Urban flooding usually takes place in unexpected events and, in consequence, useful data are not accurately recorded, as in this case. When reproducing these events it is necessary to imagine different scenarios in order to compare the relative predictions to draw conclusions. As in this work we are concerned about the accuracy of the proposed simulation model to urban flooding, we will analyze the sensitivity of the solutions to the cell size. The decrease in the cell size leads to a large increment in the time of simulation. Therefore, it is also useful to check

if good predictions can be obtained using reduced domains of the study area or if, otherwise, it is preferable to define large domains at the cost of less definition for the topographic data if extremely long computational times are to be avoided.

This hydrograph is synthetic since no actual discharge records exist [3]. As the numerical domain D_2 is located 4 km downstrean of the Tous Dam it is possible to compute a new discharge curve by recording the rate of flow discharge at an appropriate section in D_1. Due to the huge magnitude of the flooding the difference between the two discharge curves is merely a lag time of a few hours. Both are displayed in Figure 5.14. Considering this, and the fact that no records of the flood wave arrival time exist, the same original discharge curve was set as inlet boundary condition in domains D_1 and D_2 when performing numerical simulations. At the oulet boundary, downstream of the domains, the flow was let to exit freely without imposing any conditions, as no information was provided. The initial depth of water in the river reach prior to the rain events is unknown. Taking into account that the base flow of Júcar River is roughly 50 m^3s^{-1} which is totally negligible in comparison with the scale of Tous outflow hydrograph, the valley was assumed initially dry. Following [3] a Manning coefficient of $0.030sm^{-1/3}$ was used for the whole river bed reach. Other zones of increased Manning coefficient are included. As the ground in the town area was fully paved with concrete, the flood did not erode it.

Regarding recorded hydraulic data of the flooding of the town of Sumacárcel, a range for the maximum water elevation marks was collected at 21 locations within or very close to Sumacárcel village. In both calculations a total time of 39 h was simulated with a computational time of 5.5 h in the D_1 domain and 22.3 h in the D_2 domain.

These gauging points are shown in Figure 5.11. Some gauges (numbers 5, 9, 15, 17, 18 and 21) show no flooding (zero or near zero maximum water depth) and correspond to locations just barely reached by the flooding so that they represent a sort of shore line of the flood within the town.

Table 1 contains a summary of probe locations, estimated maximum water depths and computed maximum water depths on the two computational domains. The values of the water depth at gauges 1 and 2, placed in the lower part of the village indicate that the numerical solutions provided by both grids are a good prediction of the maximum water level reached by the flooding at both stations. Both gauges register almost the same water level surface evolution, as expected due to their proximity. Good agreement between maximum water elevation marks and predicted data is also found for gauge locations 3 and 4, of similar bed level elevation, and located within the village.

The results in table 1 show also a good agreement for gauge 5 that remains dry according to the field observations, despite it being close to the river bed. The elevation at gauge 6, within Sumacárcel, is overestimated in approximately 1 m. The water depth at gauge 7 agrees well with the maximum water elevation mark, whilst water depth in gauge 8 is overestimated in approximately 1 m.

Gauge	$x(m)$	$y(m)$	Est. max. $h(m)$	Comp. max. $h(m)$ D_1	Comp. max. $h(m)$ D_2
1	2410	3290	17.5-19	18.613149	18.684626
2	2400	3335	8.0-9.0	10.181195	9.806911
3	2355	3315	7.0-8.0	7.270638	7.386148
4	2345	3380	7	6.775814	6.895801
5	2335	3175	0.2	0.000	0.00
6	2335	3420	5.0-6.0	7.464109	7.615280
7	2330	3365	6	6.101556	6.143140
8	2315	3450	5	6.561674	6.679546
9	2310	3590	0	0.304004	0.119698
10	2303	3255	4	3.887516	3.979779
11	2285	3425	2	3.039008	3.194761
12	2285	3500	5.0-6.0	4.772985	4.909878
13	2280	3280	2.5-3.0	4.186196	4.330580
14	2266	3550	2	3.549098	3.122085
15	2265	3400	0	1.928118	2.134662
16	2259	3530	3.0-4.0	3.698947	3.802850
17	2250	3440	0	0.661666	0.901334
18	2230	3525	0	1.041024	1.215631
19	2205	3445	2.0-3.0	2.026697	2.257170
20	2195	3440	2	1.857008	2.096829
21	2190	3485	0	0.000	0.00

Table 5.5: Gauges position, estimated maximum water depth and simulated water depth

The results for gauges 9 and 10 show good agreement with field observations. Gauge location 9 remained dry along the flooding and the simulation provides a maximum water depth in the scale of the centimeters. The numerical results for gauge 11 indicate an overestimation of the field water depth estimation of approximately 1 m, whilst very good agreement is found for gauge 12.

The simulations at the gauge locations 13 and 14 overestimate field observations in approximately 1 m. Gauge location 15 remained dry along the flooding whereas the numerical simulation did not. On the other hand the results for gauge location 16 are in good accordance with the observed field data.

Gauges 17 and 18 remained dry but the simulation estimates a maximum depth of nearly 1 m. The results for gauge locations 19 and 20 and 21 are in accordance with field observations.

The evolution of the computed flooding can be seen in plan view in Figure 5.10 for times t =5, 10, 15, 20, 25, and 30 hours. The computed flow advances and passes around the buildings but always moving inside the limit given by that line.

Although mesh D_1 has larger cells than D_2 the numerical predictions from both grids are in general in agreement with observed data. It is remarkable that for this extreme event, despite the different locations of the inlet discharge sections and the different size of the cells in D_1 and D_2, the water depth results for D_1 are only slightly inferior than the ones obtained with D_2.

It is very useful, when an exhaustive study is required, to refine the mesh in the area of interest. In this case, the main trouble is to stablish the input hydrograph. Although water depth has no many significative differences Figure 5.12 and 5.13, velocity has not the same behaviour (Gauges 5 and 21 have been ommited because of both have calculated the dry state). In Figure 5.15 is possible to appreciate the differences where the simulation performed with the coarse mesh makes a higher estimation of the velocity.

As displayed by the results of the water level time evolution at the gauges, the mesh refinement in the zone of interest improves the quality of the predictions. The GPU simulation of the computation on the refined mesh was 22 hours and 20 minutes (more than 28 days of simulation using CPU) and that for the coarse mesh was 5 hours and 30 minutes. The coarse mesh was a good aproximation of how the flood advances but not always can be used to study the details in a particular area.

5.3 Comparing with a distributed memory parallel implementation

	28-Core$^{\diamond}$	1-Core	GPU
Cells		106648	
CFL		0.9	
t_n		400.0	
h_0		5 - 0	
Comp. Load (s.)	363.2	9383.83	250.79
S_{up}	**25.84**		**37.41**

Table 5.6: Computational load for a Dam-Break simulation (400 s.) with the mono-core version, the MPI paralellized version and the new CUDA version. $^{\diamond}$ Each core comes from an Intel i7 CPU 860 @ 2.80 GHz

This case simulates the evolution of two connected boxes where one of them contains 5 m. of water level and the other one is dry. The initial conditions and geometry are shown at 5.16.

The reason to include this additional test case is that it was run previously with a CPU version of the method paralellized through distrubuted machines paradigm using Standard MPI.

The simulation was run during 400 s. dumping data each 200 time-steps. Furthermore, it has been used CFL=0.9 and a manning coefficient of $m = 0.03$.

The results show that the power of computing of the GPU is comparable with the power of more than 30 computers working at the same time using the Distrubuted Computing paradigm. Although CUDA programming is not as easy as MPI programming and it is important to note that not every implementations support both kind of implementations, the performance of the first technique is much better.

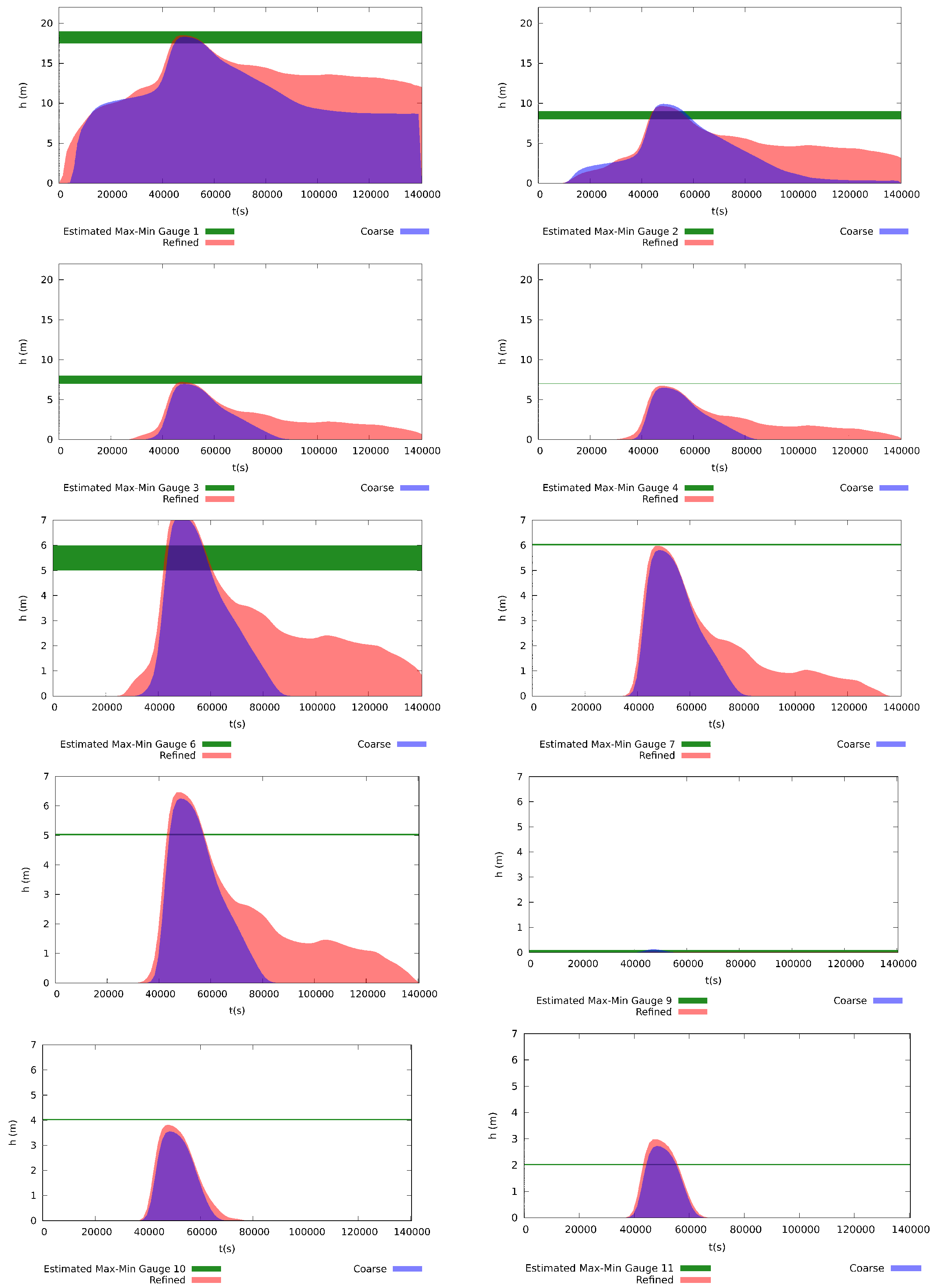


Figure 5.12: Simulated and estimated water depth in 1-11 Gauges.

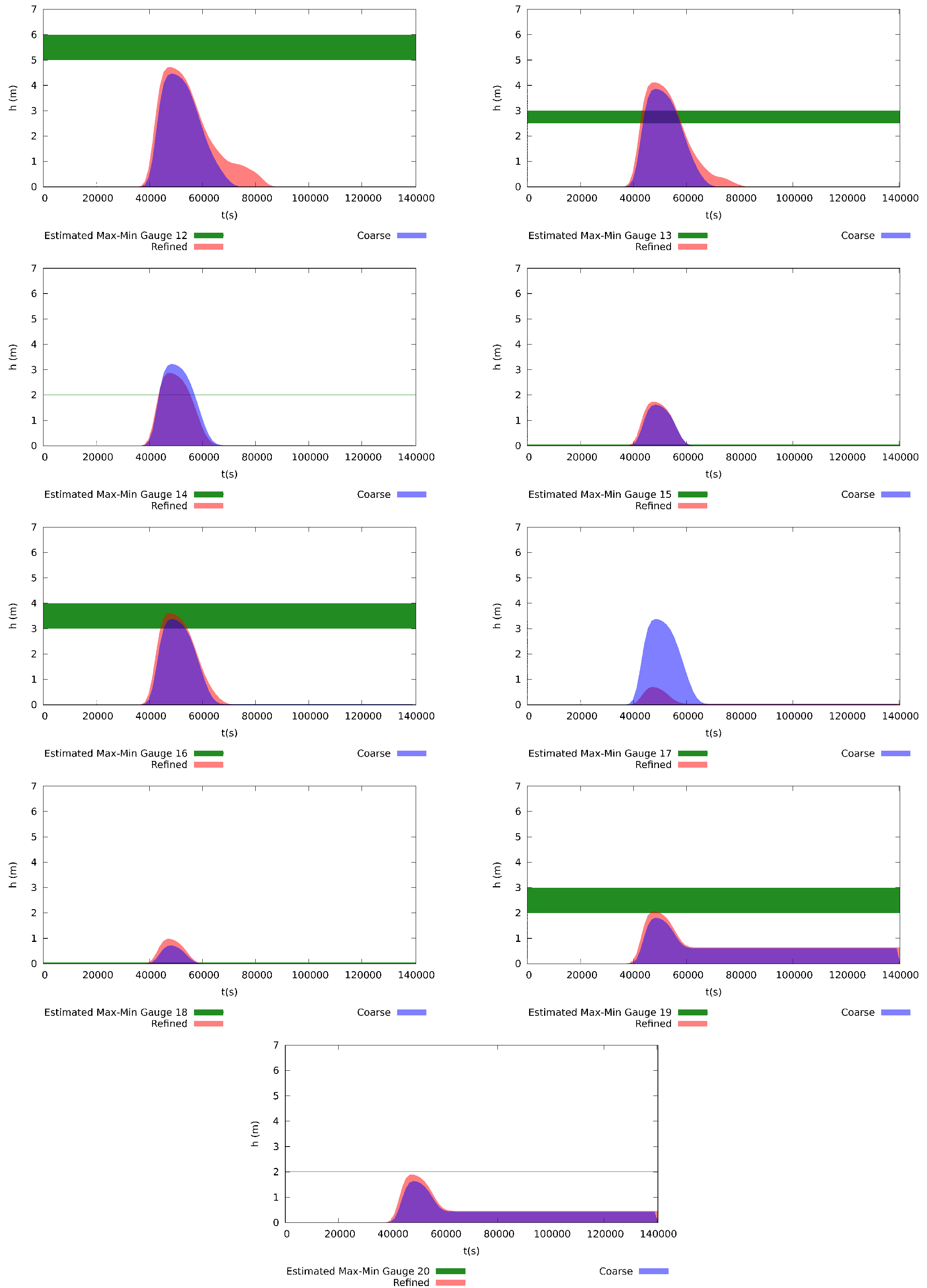


Figure 5.13: Simulated and estimated water depth in 12-21 Gauges.

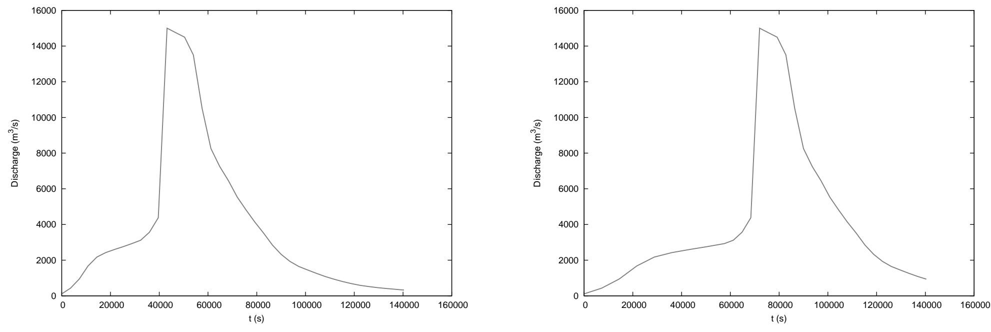


Figure 5.14: Tous synthetic hydrograph for D_1 (Right) and D_2 (Left)

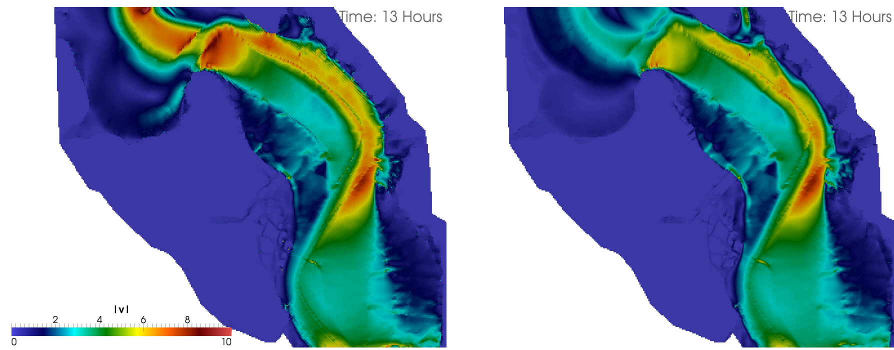


Figure 5.15: Comparison of Left: Coarse mesh velocity module and Righ: Refined mesh velocity module at $t = 13h$

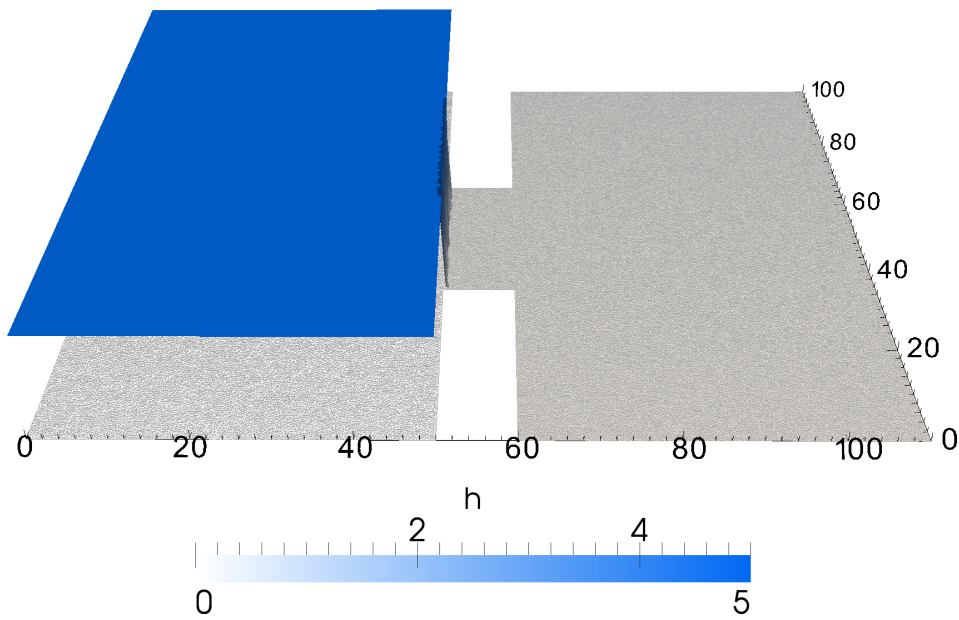


Figure 5.16: Initial conditions of water depth and mesh plot

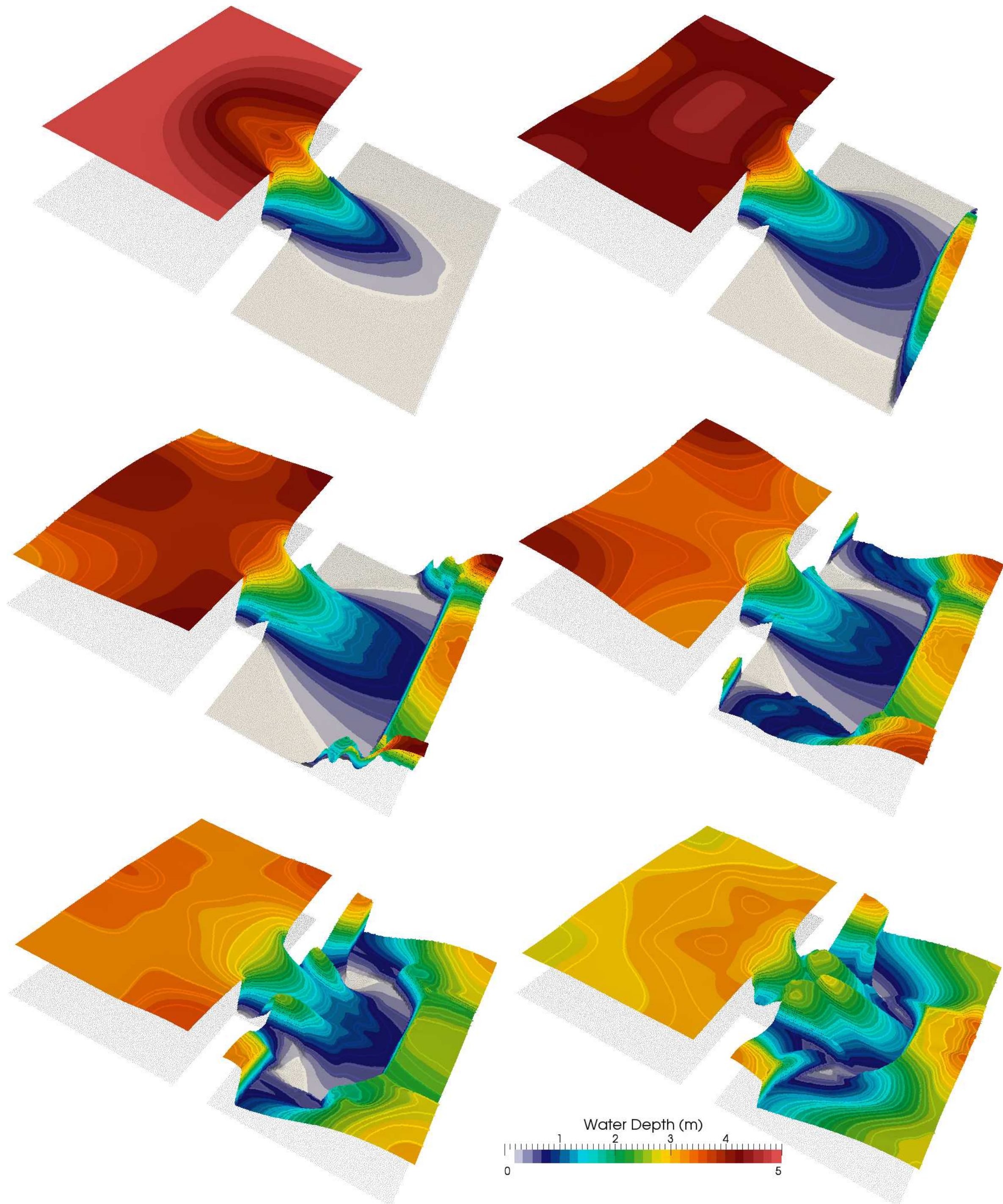
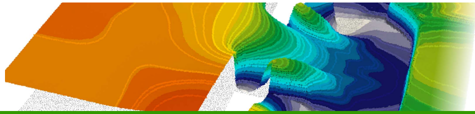


Figure 5.17: 5-0 Dam-Break simulation for (Right-Left, Top-Down) t=5, 10, 15, 20, 25, 30 seconds



6

Conclusions and future work

A first order finite volume scheme to discretize the Shallow Water equations on unstructured meshes has been implemented using GPUs. The associated speed-up has been studied when solving different problems with nVidia Tesla Series c2070. The difficulties generated by the use of unstructured meshes have been identified and partially overcome so that our results show that it is possible to solve many different problems 30 times faster than a common CPU version on a single processor. Furthermore, only machine precision differences are encountered between both implementations, so it is important to note that the speed of the simulation does not affect the precision of the numerical method.

Communicating data between CPU and GPU has a very expensive cost. An interesting strategy to reduce the impact of the communication has been proposed. The only necessity of communication is the elapsed simulation time so that the CPU schedules the operations.

Previous work related to reducing the computational cost by means of parallel CPU programming has been compared, showing that a GPU could be faster than 30 CPU cores involving less investment and less energy consumption. The values of 50-100x speed-up announced in the related literature have not been reached in our implementation. Our interpretation is that it is not possible to be more than 42 times faster than a CPU processor when working with double precision data and serious and careful speed-up comparisons are required in any case. Although it is very complicated to reach the theorical performance peak, both implementations could reach a reasonable power, so if both implementations are mostly optimized, speed ups like the related in this work are acceptable.

As further work, it is interesting to explore the Multi-GPU paradigms, simulating with many GPUs and to study other implementations which perform the memory access pattern under unstructured meshes.

Bibliography

[1] J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, and B. T. Smith. *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[2] A. A. Akanbi and N. D. Katopodes. Model for flood propagation on initially dry land. *Journal of Hydraulic Engineering*, 114(7):689–706, 1988.

[3] F. Alcrudo and J. Mulet. Description of the tous dam break case study (spain). *Journal of Hydraulic Research*, 45(sup1):45–57, 2007.

[4] A. Brodtkorb, T. Hagen, and M. Saetra. Gpu programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing (In Press)*, 2012.

[5] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar. Efficient shallow water simulations on gpus: Implementation, visualization, verification, and validation. *Computers and Fluids*, 55(0):1 – 12, 2012.

[6] J. Burguete and P. García-Navarro. Efficient construction of high-resolution tvd conservative schemes for equations with source terms: application to shallow water flows. *International Journal for Numerical Methods in Fluids*, 37(2):209–248, 2001.

[7] M. J. Castro, S. Ortega, M. de la Asunción, J. M. Mantas, and J. M. Gallardo. Gpu computing for shallow water flow simulation based on finite volume schemes. *Comptes Rendus Mécanique*, 339(2–3):165 – 184, 2011.

[8] A. Corrigan and J. Dahm. Gpu technology conference. In *Unstructured Grid Numbering Schemes for GPU Coalescing Requirement*, May 2012.

[9] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. Cpu db: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, Apr. 2012.

[10] M. Dixon, J. Chong, and K. Keutzer. Acceleration of market value-at-risk estimation. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 5:1–5:8, New York, NY, USA, 2009. ACM.

[11] P. N. Glaskowsky. Nvidia ' s fermi : The first complete gpu computing architecture. *A white paper prepared under contract with NVIDIA Corporation*, (September):1–26, 2009.

[12] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[13] M. Hubbard and P. Garcia-Navarro. Flux difference splitting and the balancing of source terms and flux gradients. *Journal of Computational Physics*, 165(1):89 – 125, 2000.

[14] A. Lacasta, P. García-Navarro, J. Burguete, and J. Murillo. Preprocess static subdomain decomposition in practical cases of 2d unsteady hydraulic simulation. *Computers and Fluids*, 2012.

[15] A. Lashgar, A. Baniasadi, and A. Khonsari. "Investigating Warp Size Impact in GPUs". *ArXiv e-prints*, may 2012.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[17] W.-Y. Liang, T.-J. Hsieh, M. T. Satria, Y.-L. Chang, J.-P. Fang, C.-C. Chen, and C.-C. Han. A gpu-based simulation of tsunami propagation and inundation. In *Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '09, pages 593–603, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] J. Murillo and P. Garcia-Navarro. Weak solutions for partial differential equations with source terms: Application to the shallow water equations. *Journal of Computational Physics Volume: 229 Issue: 11 Pages: 4327-4368*, 2010.

[19] J. Murillo and P. García-Navarro. Wave riemann description of friction terms in unsteady shallow flows: Application to water and mud/debris floods. *J. Comput. Phys.*, 231(4):1963–2001, Feb. 2012.

[20] J. Murillo, P. García-Navarro, and J. Burguete. Conservative numerical simulation of multi-component transport in two-dimensional unsteady shallow water flow. *J. Comput. Phys.*, 228(15):5539–5573, Aug. 2009.

[21] J. Murillo, P. García-Navarro, and J. Burguete. Time step restrictions for well-balanced shallow water solutions in non-zero velocity steady states. *International Journal for Numerical Methods in Fluids*, 60(12):1351–1377, 2009.

[22] J. Murillo, P. García-Navarro, J. Burguete, and P. Brufau. The influence of source terms on stability, accuracy and conservation in two-dimensional shallow flow simulation using triangular finite volumes. *International Journal for Numerical Methods in Fluids*, 54(5):543–590, 2007.

[23] nVidia Corporation. *CUDA CUBLAS Library*, Aug. 2010.

[24] P. Roe. A basis for upwind differencing of the two-dimensional unsteady euler equations. *Numerical Methods in Fluid Dynamics*, II.

[25] L. Solano-Quinde, Z. J. Wang, B. Bode, and A. K. Somani. Unstructured grid applications on gpu: performance analysis and improvement. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 13:1–13:8, New York, NY, USA, 2011. ACM.

[26] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka. Aspects of gpu for general purpose high performance computing. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ASP-DAC '09, pages 216–223, Piscataway, NJ, USA, 2009. IEEE Press.

[27] W. C. Thacker. Some exact solutions to the nonlinear shallow-water wave equations. *Journal of Fluid Mechanics*, 107:499–508, 1981.

[28] M. E. Vázquez-Cendón. Improved treatment of source terms in upwind schemes for the shallow water equations in channels with irregular geometry. *Journal of Computational Physics*, 148(2):497 – 526, 1999.

[29] Y. Wang, M. Olano, M. K. Gobbert, and W. Griffin. A GPU memory system comparison for an elliptic test problem. Technical Report HPCF–2012–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2012. (HPCF machines used: tara.).