Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture

M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, R. Doallo
Computer Architecture Group (GAC)
Univ. of A Coruña
A Coruña, Spain
{*moises.vinas, jlobeiras, basilio.fraguela, manuel.arenaz, margamor, ramon.doallo*}*@udc.es*

ABSTRACT

Shallow water simulation enables the study of problems such as dam break, river, canal and coastal hydrodynamics, as well as the transport of inert substances, such as pollutants, on a fluid. This article describes a GPU efficient and cost-effective CUDA implementation of a finite volume numerical scheme for solving pollutant transport problems in bidimensional domains. The fluid is modeled by 2D shallow water equations, while the transport of pollutant is modeled by a transport equation. The 2D domain is discretized using a first order finite volume scheme. The evaluation using a realistic problem shows that the implementation makes a good usage of the computational resources, being very efficient for real-life complex simulations. The speedup reached allowed us to complete a simulation in 2 hours in contrast with the 239 hours (10 days) required by a sequential execution in a standard CPU.

KEYWORDS: Shallow water, pollutant transport, finite volume methods, GPGPU, CUDA.

1. INTRODUCTION

Shallow water systems describe the evolution of an incompressible fluid in response to gravitational accelerations, where the depth of the layer of fluid is small compared to the dimensions of the domain. These systems have many applications, enabling the simulation of rivers, canals, coastal hydrodynamics or dam-break problems, among others. In particular, the transport of pollutant in a fluid, which is modelled by a transport equation, has particular relevance in many ecological and environmental studies. This paper uses a mathematical model that consists in the coupling of a shallow water system and a transport equation. These coupled equations constitute a hyperbolic system of conservation laws with source terms, that can be discretized using finite volume schemes [1].

Finite volume schemes solve the integral form of the shallow water equations in computational cells. Therefore, mass and momentum are conserved in each cell, even in the presence of flow discontinuities. Numerical finite volume schemes for solving the shallow water equations have been developed in many works. For example, numerical schemes for the pollutant transport problem, in the context of shallow water systems, have been developed in [2–5].

The simulations of these problems have very large computing requirements which grow with the size of the space and time dimensions of the domain. For example, in the simulation of marine systems, the spatial domain can have many kilometers and the length of the simulation time can last for months or even years. Thus, due to the interest of this kind of problems and its high computational demands, several parallel implementations have been proposed on a wide variety of platforms, such a version combining *MPI* and *SSE* (*Streaming SIMD Extensions*) instructions [6] or GPU versions [7–9]. A limitation of all these parallel implementations is that they do not handle pollutant transport problems.

This paper presents a parallel shallow water simulator implemented on a CUDA architecture that supports pollutant transport. Generic parallelizing transformations were applied to the sequential code to adapt its computational structure to the stream programming model [10], which offers great flexibility and is exploited by the programming paradigms of current *GPUs*. This paper shows that shallow water problems are well suited for the stream paradigm, and that it is possible to take advantage of the stream programming model efficiently. The resulting implementation achieves good scalability and good performance on CUDA-enabled GPUs, which enables really large simulations even when dealing with pollutant transport problems and dry-wet zones on very complex terrains.

The outline of the article is as follows. Section II describes the mathematical model of the shallow water system. Section III introduces the structure of the numerical algorithm. Section IV presents our GPU parallelization approach using *CUDA*. Section V presents experimental results for a real domain. Finally, Section VI presents conclusions and future work.

2. MATHEMATICAL MODEL: SHALLOW WATER WITH POLLUTANT TRANSPORT EQUATIONS

We use a model based on general shallow water equations coupled with a transport equation in order to simulate the transport of an inert contaminant on a fluid:

$$\left\{\begin{array}{l} \dfrac{\partial h}{\partial t} + \dfrac{\partial q_x}{\partial x} + \dfrac{\partial q_y}{\partial x} = 0 \\ \dfrac{\partial q_x}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x^2}{h} + \dfrac{1}{2}gh^2\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_x q_y}{h}\right) = gh\dfrac{\partial H}{\partial x} + ghS_{f,x} \\ \dfrac{\partial q_y}{\partial t} + \dfrac{\partial}{\partial x}\left(\dfrac{q_x q_y}{h}\right) + \dfrac{\partial}{\partial y}\left(\dfrac{q_y^2}{h} + \dfrac{1}{2}gh^2\right) = gh\dfrac{\partial H}{\partial y} + ghS_{f,y} \\ \dfrac{\partial hC}{\partial t} + \dfrac{\partial q_x C}{\partial x} + \dfrac{\partial q_y C}{\partial y} = 0 \end{array}\right. \tag{1}$$

where problem unknown variables are the water column height $h(\boldsymbol{x}, t)$, the flux $q(\boldsymbol{x}, t) = (q_x(\boldsymbol{x}, t), q_y(\boldsymbol{x}, t))$ and the pollutant concentration $C(\boldsymbol{x}, t)$. The flux is the product of the height $h(\boldsymbol{x}, t)$ and the flow speed,

$$q(\boldsymbol{x}, t) = h(\boldsymbol{x}, t)\boldsymbol{u}(\boldsymbol{x}, t) = h(\boldsymbol{x}, t)(u_x(\boldsymbol{x}, t), u_y(\boldsymbol{x}, t)), \tag{2}$$

$H(\boldsymbol{x})$ is the bottom bathymetry, i.e. the depth measured from a reference level, and therefore it doesn't depend on time.

The equation system (1) can be written as system of conservation laws with source terms:

$$\frac{\partial W}{\partial t} + \frac{\partial F_1}{\partial x}(W) + \frac{\partial F_2}{\partial y}(W) = S_1(W)\frac{\partial H}{\partial x} + S_2(W)\frac{\partial H}{\partial y} \tag{3}$$

where $q_C = hC$ and

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

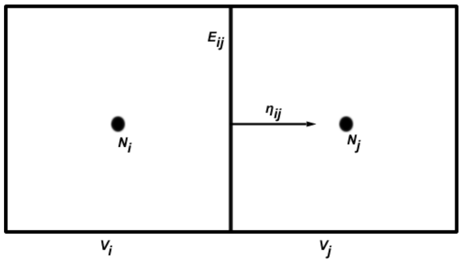


Figure 1. Finite Volume: Structured Mesh

$$W = \begin{bmatrix} h \\ q_x \\ q_y \\ q_C \end{bmatrix} \quad S_1(W) = \begin{bmatrix} 0 \\ gh \\ 0 \\ 0 \end{bmatrix} \quad S_2(W) = \begin{bmatrix} 0 \\ 0 \\ gh \\ 0 \end{bmatrix}$$

In order to discretize (3), we decompose the computational domain into cells or square control volumes [11]. As shown in 1, given a finite volume $V_i \subset \mathbb{R}^2$, $(i = 1, \ldots, L)$, $\mathcal{N}_i$ is the set of indexes having V_j and V_i as neighbors, E_{ij} is the shared common edge between them and $|E_{ij}|$ is its length, $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unitary vectorial normal to the edge E_{ij} and that points towards cell V_j.

Assuming that $W(\boldsymbol{x}, t)$ is the exact solution for system (3), we denote by ${W_i}^n$ an approximation of the average of the solution on the volume V_i and time t^n.

$$ {W_i}^n \simeq \frac{1}{|V_i|}\int_{V_i} W(\boldsymbol{x}, t^n)d\boldsymbol{x} \tag{4}$$

where $|V_i|$ is the cell's area and $t^n = t^{n-1} + \Delta t$ is the time instant, being Δt the time step.

Once the approximation W_i^n of W_i at time t^n is known, we can advance time considering a family of unidimensional Riemann problems projected in the normal direction to each edge E_{ij}. These Riemann problems can be linearized by a path-conservative Roe scheme.

Finally, the approximate solutions of these linear Riemann problems is averaged in the cells to obtain new partial constant approximations to portions of the solution. The resulting numerical scheme is as follows:

$$W_i^{n+1} = W_i^n - \frac{\Delta t}{|V_i|}\sum_{j \in \mathcal{N}_i} |E_{ij}| F_{ij}^- \tag{5}$$

with

$$F_{ij}^- = P_{ij}^-(A_{ij}(W_j^n - W_i^n) - S_{ij}(H_j - H_i)) \tag{6}$$

where $H_\alpha = H(N_\alpha), \alpha = i, j$, and A_{ij} and S_{ij} are the evaluations of

$$A(W, \boldsymbol{\eta}) = \frac{\partial F_1}{\partial W}(W)\eta_x + \frac{\partial F_2}{\partial W}(W)\eta_y \tag{7}$$

and

$$S(W,\boldsymbol{\eta}) = S_1(W)\eta_x + S_2(W)\eta_y \tag{8}$$

in $(W,\boldsymbol{\eta}) = (W_{ij},\boldsymbol{\eta}_{ij})$, being W_{ij} Roe's "intermediate state" between W_i^n and W_j^n. P_{ij} matrix is computed as follows:

$$P_{ij}^{-} = \frac{1}{2}\mathcal{K}_{ij}\cdot(I - sgn(\mathcal{D}_{ij}))\cdot\mathcal{K}_{ij}^{-1} \tag{9}$$

where I is the identity matrix, $\mathcal{D}_{ij}$ and $\mathcal{K}_{ij}$ are the eigenvalues and eigenvectors matrices, respectively.

In system (1) particular case, the average states are given by:

$$h_{ij} = \frac{h_i + h_j}{2} \tag{10}$$

$$u_{ij,\alpha} = \frac{\sqrt{h_i}\,u_{i,\alpha} + \sqrt{h_j}\,u_{j,\alpha}}{\sqrt{h_i}+\sqrt{h_j}},\ \alpha = x,y \tag{11}$$

$$C_{ij} = \frac{\sqrt{h_i}C_i + \sqrt{h_j}C_j}{\sqrt{h_i}+\sqrt{h_j}} \tag{12}$$

Jacobi matrices are given by:

$$\frac{\partial F_1}{\partial W}(W_{ij}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -u_{ij,x}^2 + c_{ij}^2 & 2u_{ij,x} & 0 & 0 \\ -u_{ij,x}u_{ij,y} & u_{ij,y} & u_{ij,x} & 0 \\ -u_{ij,x}C_{ij} & C_{ij} & 0 & u_{ij,x} \end{bmatrix} \tag{13}$$

$$\frac{\partial F_2}{\partial W}(W_{ij}) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -u_{ij,x}u_{ij,y} & u_{ij,y} & u_{ij,x} & 0 \\ -u_{ij,y}^2 + c_{ij}^2 & 0 & 2u_{ij,y} & 0 \\ -u_{ij,y}C_{ij} & 0 & C_{ij} & u_{ij,y} \end{bmatrix} \tag{14}$$

with $c_{ij} = \sqrt{gh_{ij}}$.

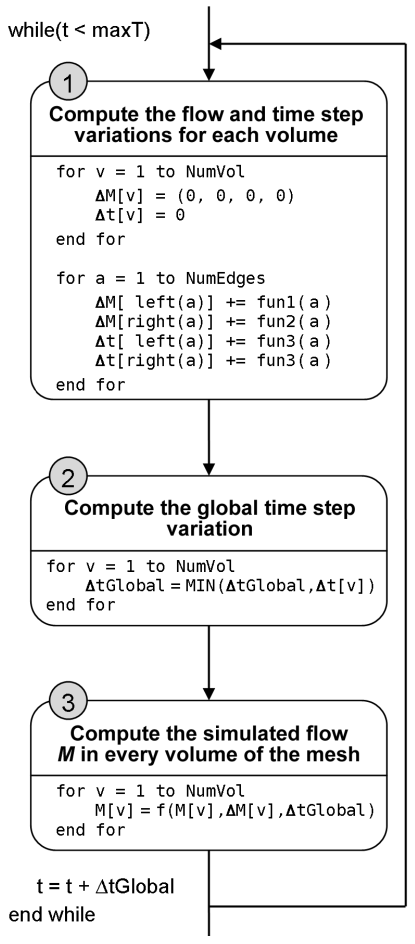
Finally, S_{ij} is given by:

$$S_{ij} = \begin{bmatrix} 0 \\ gh_{ij}\eta_{ij,x} \\ gh_{ij}\eta_{ij,y} \\ 0 \end{bmatrix} \tag{15}$$

Since the obtained numerical scheme is explicit, in order to ensure its stability, it is necessary to impose a CFL (*Courant-Friedrichs-Lewy*) condition. In practice, this condition implies a time step restriction so that to advance from iteration t^n to t^{n+1}, the time step is determined by:

$$\Delta t^n = \min_{i=1,\ldots,L}\left\{\frac{\sum_{j\in\mathcal{N}_i}|E_{ij}|\,\|D_{ij}\|_\infty}{2\gamma|V_i|}\right\} \tag{16}$$

with $\|D_{ij}\|_\infty$ being the infinity norm of matrix $\mathcal{D}_{ij}$, i.e. the maximum of matrix A_{ij} eigenvalues. As the resulting time step can be very small, it could be required to execute a large number of iterations to simulate problems that happen in large periods of time. Therefore, from the computational view point, the solution of the problem is reduced to performing a large number of small vector and matrix operations of size 4×4.


while(t < maxT)
1
Compute the flow and time step variations for each volume
2
Compute the global time step variation
3
Compute the simulated flow M in every volume of the mesh
t = t + ΔtGlobal
end while

```
for v = 1 to NumVol
    ΔM[v] = (0, 0, 0, 0)
    Δt[v] = 0
end for

for a = 1 to NumEdges
    ΔM[ left(a)] += fun1(a)
    ΔM[right(a)] += fun2(a)
    Δt[ left(a)] += fun3(a)
    Δt[right(a)] += fun3(a)
end for
```

```
for v = 1 to NumVol
    ΔtGlobal = MIN(ΔtGlobal,Δt[v])
end for
```

```
for v = 1 to NumVol
    M[v] = f(M[v],ΔM[v],ΔtGlobal)
end for
```

Figure 2. Numerical Scheme

3. STRUCTURE OF THE ALGORITHM

The algorithm that approximates the solution of the numerical scheme is shown in 2. The main loop performs the simulation through time. In each time step, the amount of flow that crosses through each edge is calculated in order to compute the flow information for every finite volume. This algorithm performs a huge number of small vector and matrix operations (product, inverse, ...) to solve the equations of the coupled system for each edge of the mesh. Each time iteration is divided in three stages:

1.- Compute the variation of the flow ΔM and the variation of time step Δt for each volume v. For each edge a, the amount of fluid and pollutant that crosses the edge towards the neighbor volume on the left, $\Delta M[left(a)]$, is computed. Furthermore, the contribution to the neighbor on the right, $\Delta M[right(a)]$, is also computed. At the beginning of this stage, $\Delta M[v]$

is set to zero ($\Delta M[v]$ has four components: the water column height, the volume flow in the x and y coordinates, and the pollutant concentration). Upon the completion of the stage, every finite volume will have received the contributions from all of its four neighbor edges. For each volume, $\Delta t[v]$ is computed in a similar manner.

2.- Compute the global time step variation $\Delta tGlobal$ as the minimum of all the time step variations stored in Δt.

3.- Compute the simulated flow M for each volume. For this purpose, the pollutant concentration and the fluid stored in $M[v]$ are updated using $\Delta M[v]$ and $\Delta tGlobal$.

Stage 1 is the most computationally intensive part. More specifically, the huge number of small vector and matrix operations involved in solving the equations are specially costly. This way, a profiling execution for an example mesh of 100×100 volumes shows that $\approx 90\%$ of the runtime is consumed by vector and matrix operations.

4. PARALLEL SOLUTION USING CUDA

Discovering the parallelism available in full-scale applications is a complex and time-consuming task. The domain-independent concept-level computational kernels recognized by the *XARK* compiler framework [12] have proved to be an useful analysis tool for whole program parallelization [13–15] and for data locality optimization [16]. Domain independent computational kernels (or simply computational kernels from now on) characterize the computations carried out in a program with independence of the programming language. Well-known examples are inductions, reductions and array recurrences. These kernels do not take into account domain-specific problem solvers. Thus, for example, a dot product and a matrix-vector product will be recognized as reduction kernels.

The sequential algorithm of Figure 2 consists of three stages that compute different types of reduction kernels: irregular reduction (Stage 1), scalar reduction (Stage 2) and regular reduction (Stage 3). The key characteristic of the reduction kernels is that the value of a memory location is updated in terms of the previous value of such memory location. A well-known example is a scalar reduction that accummulates the sum of a set of values, for instance, the result of the dot product of two vectors. The rest of this section describes our CUDA-based GPU implementation, which maps each stage to a different CUDA kernel.

The first stage consists of a loop that traverses the edges of the mesh. Each edge a writes its contribution to its two neighbor volumes $left(a)$ and $right(a)$. This procedure is an irregular reduction that is characterized by the use of an indirection array that selects the memory locations to be updated. The concurrent execution of this loop will cause write conflicts because for a given volume v the computation of $\Delta M[v]$ depends on several loop iterations (each edge of v corresponds to a different iteration). In the scope of CUDA, we rewrite the irregular reduction into a conflict free counterpart based on the recomputation of edge contributions. As shown in Figure 3, each loop iteration computes the flow associated to the four edges of a volume, so each edge is processed twice, once for each neighbor volume. Our CUDA-based implementation maps iterations to threads that run concurrently in a conflict-free manner. Although half of computations will be redundant, the great computational power of the GPUs allows to obtain a very good performance.

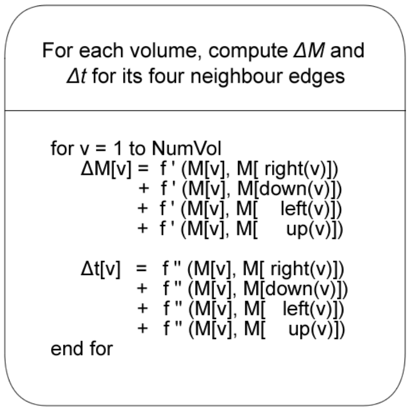
The second stage computes a scalar reduction kernel that determines the minimum time step variation $\Delta tGlobal$: CUDA supports parallel scalar reductions through a set of atomic functions. These functions use synchronization to protect write operations to shared variables by several threads. As the use of these functions decreases performance, our CUDA implementation exploits privatization and minimizes synchronization overhead. In our two-phase implementation, the set of loop iterations is partitioned among thread blocks. Each thread block runs a tree-based parallel scalar reduction in order to compute a private copy of $\Delta tGlobal$ in shared memory with coalesced read accesses to the global memory. At the end of this stage each thread block writes the value of its private copy of $\Delta tGlobal$ into the global memory. Finally, these partial results are combined to calculate the final value of the scalar reduction.

The third stage computes a regular reduction kernel, which consists of a set of scalar reductions that do not depend on each other and that may be executed in parallel in different threads. $M[v]$ and $\Delta M[v]$ are located consecutively in memory and coalesced global memory accesses are exploited.

5. EXPERIMENTAL RESULTS

The performance of our CUDA shallow water algorithm was evaluated on two CPU/GPU platforms:

System A: The first system consists of an *Intel Core 2 6600* running at 2.40 GHz, with 2 GB of DDR2 memory. It is connected to a GeForce GTX 295 GPU, with compute ca-



For each volume, compute ΔM and Δt for its four neighbour edges

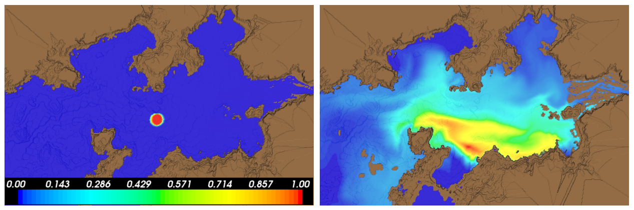
```
for v = 1 to NumVol
    ΔM[v] =  f ' (M[v], M[ right(v)])
           + f ' (M[v], M[down(v)])
           + f ' (M[v], M[   left(v)])
           + f ' (M[v], M[     up(v)])

    Δt[v]  =  f " (M[v], M[ right(v)])
           +  f " (M[v], M[down(v)])
           +  f " (M[v], M[   left(v)])
           +  f " (M[v], M[     up(v)])
end for
```

Figure 3. Stage 1 Using Recomputation-Based Solution

pability 1.3, 240 *streaming processors* (distributed in 30 *streaming multiprocessors*, each one having 8 streaming processors) and 896 MB of GDDR3 memory. This GPU reaches a memory bandwidth of 112 GB/s. The software setup is *Debian GNU/Linux 6.0 (squeeze)* operating system using *g++ 4.3.5* compiler and *nvcc* version 3.2 to compile CUDA code. The GPU version was compiled with g++ because nowadays CUDA is not compatible with the 12.x version of the icpc compiler.

System B: The second system has an *Intel Xeon X5650* running at 2.67 GHz, with 6 cores and 2 threads by core reaching a maximum memory bandwidth of 32 GB/s [17]. This CPU has 12 GB of DDR3 memory. It is connected to an S2050 GPU with *Fermi* architecture, 448 *streaming processors* (distributed in 14 *streaming multiprocessors*, each one having 32 streaming processors), with 3 GB of GDDR5 memory. This GPU provides 1030 GFlops of performance in single precision floating point reaching a memory bandwidth (with ECC disabled) of 150 GB/s. This is a device of compute capability 2.X thus, we can configure its L1 cache size vs the shared memory size. In our case, we have configured the S2050 GPU with 48 KB of L1 cache and 16 KB of shared memory using `cudaFuncSetCacheConfig()` [18]. This choice is recommended in [18] for kernels that use a lot of local memory, which is in fact the case in the kernels developed for this application. The reason of good performance of choosing this option is because of the local memory and the global memory are competing for the L1 cache and incrementing the amount of L1 cache provides more cache size for both memory types. The software setup is *Debian GNU/Linux 6.0.1 (squeeze)* operating system using *icpc 12.0.2* and *g++ 4.3.5* compilers and *nvcc* version 3.2.

The natural environment is the *Ría de Arousa*, an estuary in Galicia (Spain). In this test this environment is simulated using real terrain and bathymetry data. The north and east limits have free boundary conditions, while in the south and west borders the tides are simulated using barometric tidal equations. As shown in Figure 4(a), a discharge of pollutant is artificially added in order to study its propagation and determine the most affected areas. The total simulated period is 604800 seconds (one week of real time) (see Figure 4(b)).



(a) Initial setup (b) Pollutant concentration after eight days

Figure 4. Development Environment

Table 1. Execution Time (in Seconds) for the System A

Mesh size	*CPU/Seq*	*CPU/OpenMP*		*GPU GTX 295*	
	time	time	speedup	time	speedup
100 × 100	2654	1335	1.99x	66.64	39.8x
200 × 200	21086	10629	1.98x	239.66	88.0x
300 × 300	69560	35702	1.95x	549.80	126.5x
400 × 400	164850	84234	1.96x	1059.14	155.6x
500 × 500	319858	163736	1.95x	1905.45	167.9x
600 × 600	N-A	N-A	N-A	3083.03	N-A
700 × 700	N-A	N-A	N-A	4777.12	N-A
800 × 800	N-A	N-A	N-A	6817.04	N-A
900 × 900	N-A	N-A	N-A	9689.67	N-A
1000 × 1000	N-A	N-A	N-A	12833.41	N-A

Tables 1 and 2 show the execution time of the sequential implementation (*CPU/Seq*) of several mesh sizes in the System A and the System B, respectively. The tables also include the execution times of a parallel OpenMP-based CPU implementation (*CPU/OpenMP*) and a parallel GPU implementation (*GPU GTX 295* for the System A and *GPU S2050* for the System B). The corresponding speedups are computed with respect to the *CPU/Seq* sequential time of its system. For the System A, we have only obtained the execution time for the smaller grids because for the bigger meshes the execution times were too high.

The GPU speedup column of Table 1 shows that the GPU implementation reaches a speedup of 167.9x in the System A. For example, for the 100×100 grid in the Intel Core 2 CPU the sequential code takes almost 44 minutes while the CUDA code takes only 66 seconds. For the 400×400 grid, the sequential code requires almost 46 hours and the CUDA code takes 18 minutes for GTX 295 GPU. The results for the System B depicted in Table 2 reflect the performance difference between the two systems. For example, for the

Table 2. Execution Time (in Seconds) for the System B

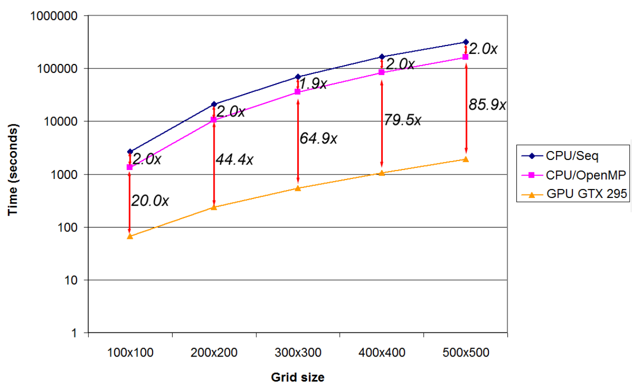
Mesh size	*CPU/Seq*	*CPU/OpenMP*		*GPU S2050*	
	time	time	speedup	time	speedup
100 × 100	932	157	5.9x	17.53	53.2x
200 × 200	7440	1086	6.8x	80.34	92.6x
300 × 300	23912	3443	6.9x	233.06	102.6x
400 × 400	56256	8361	6.7x	505.12	111.4x
500 × 500	109201	16527	6.6x	980.64	111.4x
600 × 600	188125	28580	6.6x	1624.35	115.8x
700 × 700	297849	45344	6.6x	2606.61	114.3x
800 × 800	443246	67110	6.6x	3779.21	117.3x
900 × 900	629210	95461	6.6x	5468.73	115.1x
1000 × 1000	860457	130815	6.6x	7315.71	117.6x

400 × 400 grid, in this CPU the sequential code takes just over 15 hours and a half, 2 hours and 20 minutes for the OpenMP version and almost 8 minutes for GPU version.

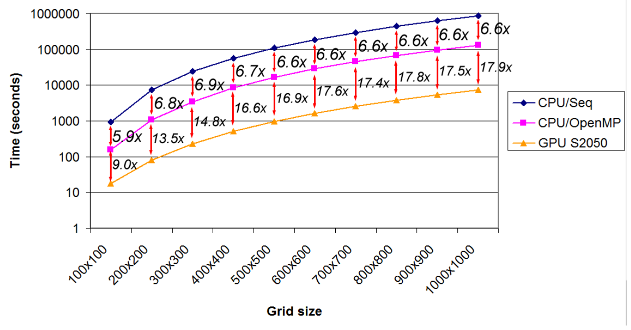
It is very remarkable that in some cases, the execution time is shorter than the simulated time and therefore we can work in real time. For example, for the biggest mesh (1000 × 1000), the System B has a sequential simulation time of 10 days (this is, three days more that the simulated time) while the *GPU GTX 295* only takes 2 hours.

Figure 5 shows a graphical representation of the execution times using a logaritmic scale. We see that the GPU speedup grows gradually with the problem size. One reason is that small grids fit in the cache hierarchy of the CPU. Therefore, the bigger the grid size, the more accesses (by iteration) to the main memory of the CPU. Another reason for the low speedups in small grids, is that small grids lead to the creation of fewer thread blocks than bigger grids. Thus, when a thread block accesses to global memory (because of a global memory access, texture cache miss or local memory access), the thread scheduler chooses other thread block ready to execute. If the number of thread blocks is low, it is possible that there are not any thread blocks ready to execute. For example, for the 100 × 100 grid, we have 12 thread blocks by streaming multiprocessor. However for 1000 × 1000 grid, we have 1117 thread blocks; that is almost 100 times more thread blocks.

From the point of view of power consumption, our results point that GPUs are a better solution to parallelize this kind of problems than CPUs. For example, the system conformed by the Intel Core 2 (65 W of power consumption) and the GeForce GTX 295 (289 W of maximum power consumption) is 86 times faster than the (*CPU/OpenMP*) multithread solution executed on Intel Core 2 (65 W of power consumption). However, the power consumption of one Intel Core 2 is only 5 times shorter than the Core 2 + GTX 295 group. The ratio power consumption-speedup is clearly favorable to the GPU. In other words, with perfect scaling 86 CPUs would be necessary to reach that speedup, which



(a) System A



(b) System B

Figure 5. Execution Time for Several Mesh Sizes and Speedups between *CPU/Seq* and *CPU/OpenMP* and, *CPU/OpenMP* and *GPU*

would obviously require 86 times the power of one CPU.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have solved the simulation of pollutant transport in shallow water systems with the Nvidia CUDA technology. The problem presented has great interest in many industrial and environmental projects, being very computationally costly.

For the two systems, we can ensure that it would be possible the use in real time of this CUDA version of the shallow water simulator to follow the evolution of a pollutant in real environments. As the simulated time is seven days, the simulation can be done in real time if the simulation time is shorter than seven days. This fact occurs for the System B, the *CPU/OpenMP* takes 36 hours (1.5 days); for the System A, the *GPU GTX 295* takes 4 hours and; for the System B, the *GPU S2050* takes only 2 hours of simulation to simulate the environment during seven days. As future work we will consider the implementation of a multi-GPU solution and changing the recomputation-based solution by a more efficient one.

ACKNOWLEDGMENTS

We would like to thank the G-HPC network for promoting interdisciplinary collaborations between groups of the network. This research has been supported by the Xunta de Galicia under projects INCITE08PXIB105161PR and 08TIC001206PR, the Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union under the grant TIN2010-16735, and the Consolidation of Competitive Research Groups ref. 2010/06. Finally, we also thank the Consellería do Mar of Xunta de Galicia (local government of Galicia, Spain) and the Centro Tecnolóxico do Mar (CETMAR) for providing the ocean currents and topographic data of Ría de Arousa.

REFERENCES

[1] R.J. LeVeque, FINITE VOLUME METHODS FOR HYPERBOLIC PROBLEMS. Cambridge University Press, 2002.

[2] M.-O. Bristeau, B. Perthame, "Transport of pollutant in shallow water using kinetic schemes," *ESAIM, CEMRACS 1999*, vol. 10, pp. 9–21, 2001.

[3] Z. Xu, C. Shu, "Anti-diffusive finite difference WENO methods for shallow water with transport of pollutant," *J. Comput. Math.*, vol. 24, no. 3, pp. 239–251, 2006.

[4] E. Audusse, M.-O. Bristeau, "Transport of pollutant in shallow water. A two time steps kinetic method," *ESAIM: M2AN*, vol. 37, no. 2, pp. 389–416, 2003.

[5] F. Benkhaldoun, I. Elmahi, M. Seaïd, "Well-balanced finite volume schemes for pollutant transport on unstructured meshes," *Journal of Computational Physics*, vol. 226, no. 1, pp. 180–203, 2007.

[6] M.J. Castro, J.A. García-Rodríguez, J.M. González-Vida, C. Parés, "Solving shallow-water systems in 2D domains using finite volume methods and multimedia SSE instructions," *J. Comput. Appl. Math.*, vol. 221, no. 1, pp. 16–32, 2008.

[7] M.J. Castro, S. Ortega, M. de la Asunción, J.M. Mantas, J.M. Gallardo, "GPU computing for shallow water flows simulation based on finite volume schemes," *Comptes Rendus Mécanique*, vol. 339, no. 2-3, pp. 165–184, 2011.

[8] M. de la Asunción, J.M. Mantas, M.J. Castro, "Programming CUDA-based GPUs to simulate two-layer shallow water flows," *Lecture Notes in Computer Science. Euro-Par 2010. Parallel Processing*, pp. 353–364, 2010.

[9] J.M. Gallardo, S. Ortega, M. de la Asunción, J.M. Mantas, "Two-dimensional compact third-order polynomial reconstructions. Solving nonconservative hyperbolic systems using GPUs," *Journal of Scientific Computing*, pp. 1–23, 2011.

[10] J. Gummaraju, M. Rosemblum, "Stream programming on general-purpose processors," in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 343–354, IEEE Computer Society, 2005.

[11] M.J. Castro, E.D. Fernández-Nieto, A.M. Ferreiro, J.A. García-Rodríguez, C. Parés, "High order extensions of ROE schemes for two dimensional nonconservative hyperbolic systems," *Journal of Scientific Computing*, vol. 39, no. 1, pp. 67–114, 2009.

[12] M. Arenaz, J. Touriño, R. Doallo, "XARK: An eXtensible framework for Automatic Recognition of computational Kernels," *ACM Transactions on Programming Languages and Systems*, vol. 30, no. 6, pp. 1–56, 2008.

[13] M. Arenaz, J. Touriño, R. Doallo, "Compiler support for parallel code generation through kernel recognition," in *Proc. of the 18th IEEE International Parallel and Distributed Processing Symposium*, pp. 79b (CD–ROM, 10 pages), 2004.

[14] J. Setoain, C. Tenllado, J.I. Gómez, M. Arenaz, M. Prieto, J. Touriño, "Towards automatic code generation for GPU architecture," in *Proc. of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Comput.*, 2008.

[15] J. Lobeiras, M. Amor, M. Arenaz, B.B. Fraguela, J.A. García, M.J. Castro, "Simulación de aguas poco profundas en una GPU mediante Brook+," in *Proc. of XX Jornadas de Paralelismo*, pp. 327–332, 2009. (in Spanish).

[16] D. Andrade, M. Arenaz, B.B. Fraguela, J. Touriño, R. Doallo, "Automated and accurate cache behavior analysis for codes with irregular access patterns," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2407–2423, 2007.

[17] Intel, "Intel Xeon X5650 Processor."

[18] Nvidia, *Tuning CUDA Applications for Fermi*, 1.3 ed., August 2010.