CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–16
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Multi-GPU Shallow Water Simulation with Transport of Contaminants

M. Viñas[1], J. Lobeiras[1], B.B Fraguela[1], M. Arenaz[1], M. Amor[1], J.A. García[2], M.J. Castro[3] and R. Doallo[1]

[1] *Grupo de arquitectura de computadores (GAC), Univ. de A Coruña (UDC)*
[2] *Grupo M2NICA, Univ. de A Coruña (UDC)*
[3] *Grupo EDANYA, Univ. de Málaga (UMA)*

SUMMARY

This work presents cost-effective multi-GPU parallel implementations of a finite volume numerical scheme for solving pollutant transport problems in bidimensional domains. The fluid is modelled by 2D shallow water equations, while the transport of pollutant is modelled by a transport equation. The 2D domain is discretized using a first order Roe finite volume scheme. Specifically, this paper presents *multi-GPU* implementations of both a solution that exploits recomputation on the GPU, and an optimized solution that is based on a ghost cell decoupling approach. Our multi-GPU implementations have been optimized using nonblocking communications, overlapping communications and computations, and applying ghost cell expansion in order to minimize communications. The fastest one reached a speedup of 78x using 4 GPUs on an Infiniband network with respect to a parallel execution on a multicore CPU with 6 cores and 2-way hyperthreading per core. Such performance, measured using a realistic problem, enabled the calculation of solutions not only in real-time, but orders of magnitude faster than the simulated time.

Received . . .

KEY WORDS: Shallow water; pollutant transport; finite volume methods; CUDA; multi-GPU; recomputation; ghost cell decoupling

1. INTRODUCTION

Shallow water systems are commonly used to simulate the behavior of a fluid when the height of fluid is small when compared to the horizontal dimensions of the studied domain. Thus, these systems can be used to simulate river and coastal currents, among other applications. In many situations, the fluid may transport a pollutant. In such situations, an extra equation is added to model the transport phenomena. The coupled system has relevance in many ecological and enviromental studies. From the mathematical point of view, the resulting coupled system constitutes a hyperbolic system of conservation laws with source terms, which 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 (see for example [2] and the references there in). Numerical schemes for

*Correspondence to: [1] {moises.vinas, jlobeiras, basilio.fraguela, manuel.arenaz, margamor, ramon.doallo}@udc.es
[2] jagrodriguez@udc.es
[3] castro@anamat.cie.uma.es

the pollutant transport problem, in the context of shallow water systems, have been developed in [2, 3, 4, 5, 6].

The simulation of these problems may have heavy computational requirements. For instance, the simulation of tidal currents in a marine basin is usually carried out in big spatial domains (up to many kilometers), and during long periods of time (several months or even years). Due to the interest of this kind of problems and their high computational demands, several parallel implementations have been proposed on a wide variety of platforms, such as a version combining *MPI* (*Message Passing Interface*) and *SSE* (*Streaming SIMD Extensions*) instructions [7], single-GPU versions [8, 9] or CUDA-based multi-GPU solutions [10, 11]. In [10] an efficient implementation of a shallow water system that overlaps computation with communication reaches almost a perfect scaling on a GPU cluster of 32 nodes but it does not use the Ghost Cell Expansion technique [17] to reduce the inter-node communication frequency. In our work, we compare two different multi-GPU implementations that overlap computation with communication, and that reduce the communication frequency. In [11] a shallow water solver is presented and tested in a single node with four GPUs reaching near-perfect weak and strong scalling. In this work, we also use four GPUs but they are divided into two nodes, which requires to use message passing on a network between several MPI processes. Finally, note that both [10, 11] do not consider the pollutant transport problem.

The main objective of this work is to present a CUDA-based multi-GPU parallel shallow water simulator that supports pollutant transport as well as dry-wet fronts in emerging bottom situations. The starting point consists of two different single-GPU solutions: first, a naive solution that exploits recomputation on the GPU; and second, an optimized solution that is based on ghost cell decoupling, on the efficient use of the GPU shared memory to minimize global memory accesses, and on the use of the texture memory to optimize uncoalesced global memory accesses. The paper presents multi-GPU versions of these naive and optimized single-GPU solutions. They use nonblocking communications to overlap communication with computation and the Ghost Cell Expansion technique to minimize the communication frequency. Overall, this paper shows that shallow water problems are well suited for exploiting the stream programming model on multi-GPU systems. The resulting implementations achieve excellent performance on CUDA-enabled GPUs and make efficient usage of our multi-GPU system, which makes feasible the execution of 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 2 describes the shallow water underlying mathematical model. Section 3 introduces the naive single-GPU CUDA implementation based on recomputation. Section 4 presents the optimized single-GPU implementation based on ghost cell decoupling. Section 5 details the implementation of the two multi-GPU versions contributed in this paper. Section 6 discusses the experimental results. Finally, Section 7 presents the conclusions.

2. MATHEMATICAL MODEL: SHALLOW WATER WITH POLLUTANT TRANSPORT EQUATIONS

A pollutant transport model consists in the coupling of a fluid model and a transport equation. Here, the bidimensional shallow water system is used to model the hydrodynamical component and a single transport equation is added to complete the system:

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

DOI: 10.1002/cpe

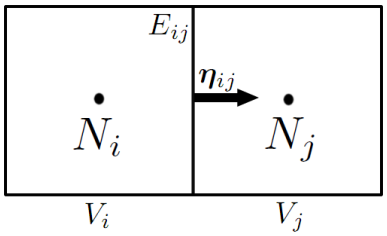


Figure 1. Finite volume: structured mesh

where problem unknowns are the water column height $h(\boldsymbol{x},t)$, the vertical averaged flux $q(\boldsymbol{x},t) = (q_x(\boldsymbol{x},t), q_y(\boldsymbol{x},t))$ and the vertical averaged pollutant concentration $C(\boldsymbol{x},t)$. The mean velocity field is related with the flux by the equation,

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

g is the gravity and $H(\boldsymbol{x})$ is the bottom bathymetry measured from a reference level, and we suppose that it does not depend on time. Here, we have neglected the friction terms, although they play an important role in practical applications.

The system (1) can be written as a 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

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

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

being $q_C = hC$.

In order to discretize (3), the computational domain is divided into cells. In this work we use Cartesian structured grids. The following notation is used (see Figure 1): given a finite volume $V_i \subset \mathbb{R}^2$, $(i = 1, \ldots, L)$, N_i is its geometrical center, $\mathcal{N}_i$ is the set of indexes j, such that V_j is a neighbor of V_i, E_{ij} is the shared common edge between them and $|E_{ij}|$ its length, and $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unit normal vector to E_{ij} pointing 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.

Let us suppose that W_i^n is known, then to advance in time, a family of unidimensional Riemann problems projected in the normal direction to each edge E_{ij} are considered. These Riemann problems can be linearized by a path-conservative Roe scheme. Finally, W_i^{n+1} is computed by averaging the solutions of each Riemann problem at each cell. The resulting numerical scheme is as follows:

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

DOI: 10.1002/cpe

with $\Delta t = t^{n+1} - t^n$ the time step, and

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

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, respectively, the matrix of eigenvalues and eigenvectors of A_{ij}.

The Roe state for system (1) is given by $W_{ij} = [h_{ij},\ h_{ij}u_{ij,x},\ h_{ij}u_{ij,y}\ h_{ij}C_{ij}]$:

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

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

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

Jacobian 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:

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

In order to ensure the stability of the explicit numerical scheme previously presented, it is necessary to impose a CFL (*Courant-Friedrichs-Lewy*) condition. In practice, this condition implies a time step restriction:

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

where $\|D_{ij}\|_\infty$ is the maximum of the absolute values of the eigenvalues of matrix A_{ij} and $\gamma \leq 1$. Note that, in practice, the resulting time step may be very small, so a huge amount of time steps may be performed to obtain the final simulation. From the computational point of view, a big amount of small vector and matrix operations of size 4×4 should be performed in each time step.

DOI: 10.1002/cpe

```
1   t = 0;
2   while t < simulation_time do
3         /* Stage1 */
4         for all v do
5               ΔM[v] = f' (M[v], M[right(v)])
6                     + f' (M[v], M[down (v)])
7                     + f' (M[v], M[left (v)])
8                     + f' (M[v], M[up (v)])
9
10              Δt[v] = f''(M[v], M[right(v)])
11                    + f''(M[v], M[down (v)])
12                    + f''(M[v], M[left (v)])
13                    + f''(M[v], M[up (v)])
14        end for
15
16        /* Stage2 */
17        for all v do
18              ΔtGlobal = MIN(Δt[v])
19        end for
20
21        /* Stage3 */
22        for all v do
23              M[v] = f (M[v], ΔM[v], ΔtGlobal)
24        end for
25
26        t = t + ΔtGlobal
27  end while
```

Figure 2. Naive algorithm

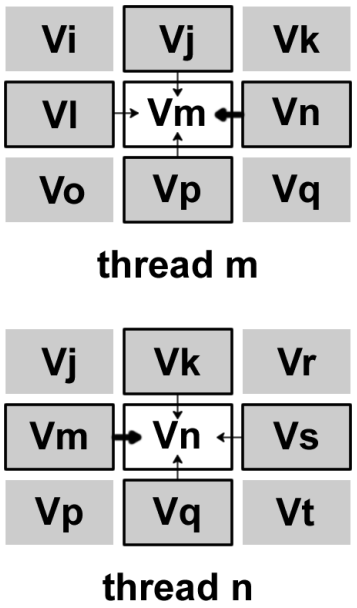


Figure 3. Recomputation-based solution on a multithreading system

3. NAIVE SINGLE-GPU SOLUTION

This section explains a naive single-GPU solution that uses a recomputation-based algorithm to take advantage of the computational power of the GPU. It is developed using only the basic features of CUDA programming and avoiding hardware-dependent tuning techniques. Figure 2 shows the algorithm corresponding to the numerical scheme of the coupled system given by Equation 3. 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 of data for all finite volumes. This algorithm performs a huge number of small vector and matrix operations to solve the equations of the coupled system for each edge of the mesh. Each time iteration is divided into 3 stages:

Stage1. Computation of the flow of data ΔM and the time step Δt for each volume v (lines 4-14 in Figure 2) applying a recomputation-based solution. For each volume, the recomputation-based algorithm calculates four flow contributions (up, down, left and right) that are associated to each of the four edges of a volume. This implies that each edge is processed twice, once for each neighbor volume. For each volume, $\Delta t[v]$ is computed in a similar way. Our naive CUDA implementation maps volumes to threads that run concurrently in a conflict-free manner. Although one half of the computations will be redundant, the great computational power of the GPUs allows to obtain a competitive performance. Figure 3 depicts an example for two threads. The contribution that volume Vm does to volume Vn takes the same value (though opposite sign) as the contribution of volume Vn to volume Vm. However this contribution is recalculated when volume Vm computes the contribution from its neighbors. This first kernel only performs accesses to global memory and thus, it saves arrays ΔM and Δt in global memory.

Stage2. Computation of the global time step $\Delta tGlobal$ (lines 17-19 in Figure 2) as the minimum of the local time steps Δt computed for each volume in *Stage1*. CUDA supports atomic operations on global memory, but we do not use them because their performance is very poor in practice. The implementation of *Stage2* is based on a reduction kernel that is launched many times in

DOI: 10.1002/cpe

order to reduce the array Δt allocated in global memory. In each invocation of this reduction kernel, the set of loop iterations is partitioned among thread blocks so that read accesses to global memory are coalesced. Each thread block runs a tree-based parallel reduction that operates only on a buffer allocated in shared memory. The partial result is saved in a private copy of $\Delta tGlobal$ allocated in shared memory. At the end of the kernel invocation, each thread block writes this partial result into a different element of array $\Delta tGlobal$ allocated in global memory. Finally, when the size of the array is twice the thread block size, no more reduction kernels are launched and this array is reduced by the CPU.

Stage3. Computation of the simulated flow data M for each volume (lines 22-24 in Figure 2). This is achieved by updating in each volume the pollutant density and the fluid data using ΔM from *Stage1* and $\Delta tGlobal$ from *Stage2*. This stage computes a set of operations that do not depend on each other and that, therefore, will be executed in parallel in different threads.

The naive single-GPU implementation described above differs from the solution presented in [12], where a reduction kernel based on `reduce3` of the CUDA SDK [13] is used in the *Stage2*. In this work, we use a kernel based on the `reduce5` implementation of the CUDA SDK. This kernel is completely unrolled, and avoids divergence, shared memory bank conflicts and unnecessary synchronization points.

The first stage is the most computationally intensive part of the algorithm, being the huge number of small vector and matrix operations needed to solve the equations specially costly. This way, a profiling execution for an example mesh of 1000×1000 volumes shows that about 80% of the runtime is consumed by the computations done in this first stage.

4. OPTIMIZED SINGLE-GPU SOLUTION

In this section, an efficient single-GPU implementation based on the ghost cell decoupling technique is proposed. This implementation, whose structure is shown in Figure 4, contains three improvements with respect to the naive implementation presented in Section 3. The first improvement (see lines 7-22 in Figure 4) is the application of a ghost cell decoupling technique to *Stage1* in order to avoid most of the duplicated computations of the recomputation-based solution. Our ghost cell decoupling strategy uses shared memory to save the local time steps before storing them into global memory. This leads naturally to the second improvement (see lines 24-34 in Figure 4), which consists in splitting the reduction of *Stage2* into two phases: first, each thread block of the kernel of *Stage1* reduces its local time steps in shared memory and saves partial results in global memory (see lines 24-28); and second, the kernel of *Stage2* reduces the partial results using the `reduce5` CUDA implementation (see lines 31-34). The third improvement is the usage of texture memory when uncoalesced memory accesses occur, provided that the arrays affected by those accesses do not change during the execution and the consistency of the texture memory can be guaranteed. This avoids the time penalties of uncoalesced accesses to global memory. The rest of this section describes these three improvements in more detail. Their impact on the execution time will be studied in Section 6.

4.1. Ghost cell decoupling solution

This improvement is aimed to reduce the large number of duplicated computations that arise in the recomputation-based solution used in *Stage1* (lines 4-14 of Figure 2). This improvement starts with a decomposition of the 2D domain using the ghost cell decoupling technique. This technique enables a memory conflict-free execution of the thread blocks (avoiding communications and synchronization between thread blocks). For this purpose, the 2D domain is splitted into 2D subdomains that include several ghost cells. The ghost cells represent flux contributions that are recomputed in two neighbor thread blocks. In our shallow water problem, these ghost cells are a row and a column of each 2D subdomain. This way, this memory region (*ghost region* from now on) is read by two thread blocks, although it is only updated by one. The reason is that the ghost region, together with the rows and columns whose updating is responsability of the thread block, provide the information the

DOI: 10.1002/cpe

```
1  t = 0;
2  blocks;  /*The number of thread blocks of the first kernel*/
3  while t < simulation_time do
4
5        /* Stage1 */
6        for block ∈ {0 . . . blocks} do
7            for all v ∈ block do
8                flR[v] = f' (M[v], M[right(v)])
9                flD[v] = f' (M[v], M[ down(v)])
10               ΔM[v] = flR[v] + flD[v]
11
12               dtR[v] = f''(M[v], M[right(v)])
13               dtD[v] = f''(M[v], M[ down(v)])
14               Δt[v] = dtR[v] + dtD[v]
15           end for
16
17           sync_barrier
18
19           for all v ∉ {GHOST_REGION} do
20               ΔM[v] = ΔM[v] - flR[left(v)] - flD[ up(v)]
21               Δt[v] = Δt[v] + dtR[left(v)] + dtD[ up(v)]
22           end for
23
24           sync_barrier
25
26           for all v ∉ {GHOST_REGION} do
27               Δt[block] = MIN(Δt[v])
28           end for
29       end for
30
31       /* Stage2 */
32       for all block ∈ {0 . . . blocks} do
33           ΔtGlobal = MIN(Δt[block])
34       end for
35
36       /* Stage3 */
37       for all v do
38           M[v] = f (M[v], ΔM[v], ΔtGlobal)
39       end for
40
41       t = t + ΔtGlobal
42 end while
```

Figure 4. Optimized GPU solution

thread block needs to perform its computations, making therefore the block self-sufficient. Overall, the ghost cell decoupling technique removes the replicated computations for most of the cells, the exception being the ghost cells of each thread block.

The algorithm shown in Figure 4 shows the implementation details of the ghost cell decoupling technique. The thread responsible for volume v computes the flow from the neighbor volumes on the right (flR in line 8) and bottom (flD in line 9). Next, a partial flow $\Delta M[v]$ is calculated as $flR[v] + flD[v]$ (line 10). The same procedure is followed to obtain the partial $\Delta t[v]$ (lines 12-14). These partial values $flR[v]$, $flD[v]$, $dtR[v]$ and $dtD[v]$ are stored in the shared memory, so that in a second phase (lines 19-22) the thread responsible for volume *v* only has to add to its partial flow the opposite contribution of its left and up neighbors (see line 20; correspondingly see $\Delta t[v]$ in line 21). A synchronization barrier is needed between the first and the second phase (line 17) because in the second phase each thread reads the partial flows (lines 20-21) stored in shared memory by another thread in the first phase (lines 8-14). In CUDA, a synchronization barrier (`__syncthreads()`) stops all warps within a given thread block until all the warps have reached the synchronization barrier. This way, the synchronization barrier guarantees that all threads of the thread block have stored their partial flows and timesteps in shared memory before another thread makes use of them.

DOI: 10.1002/cpe

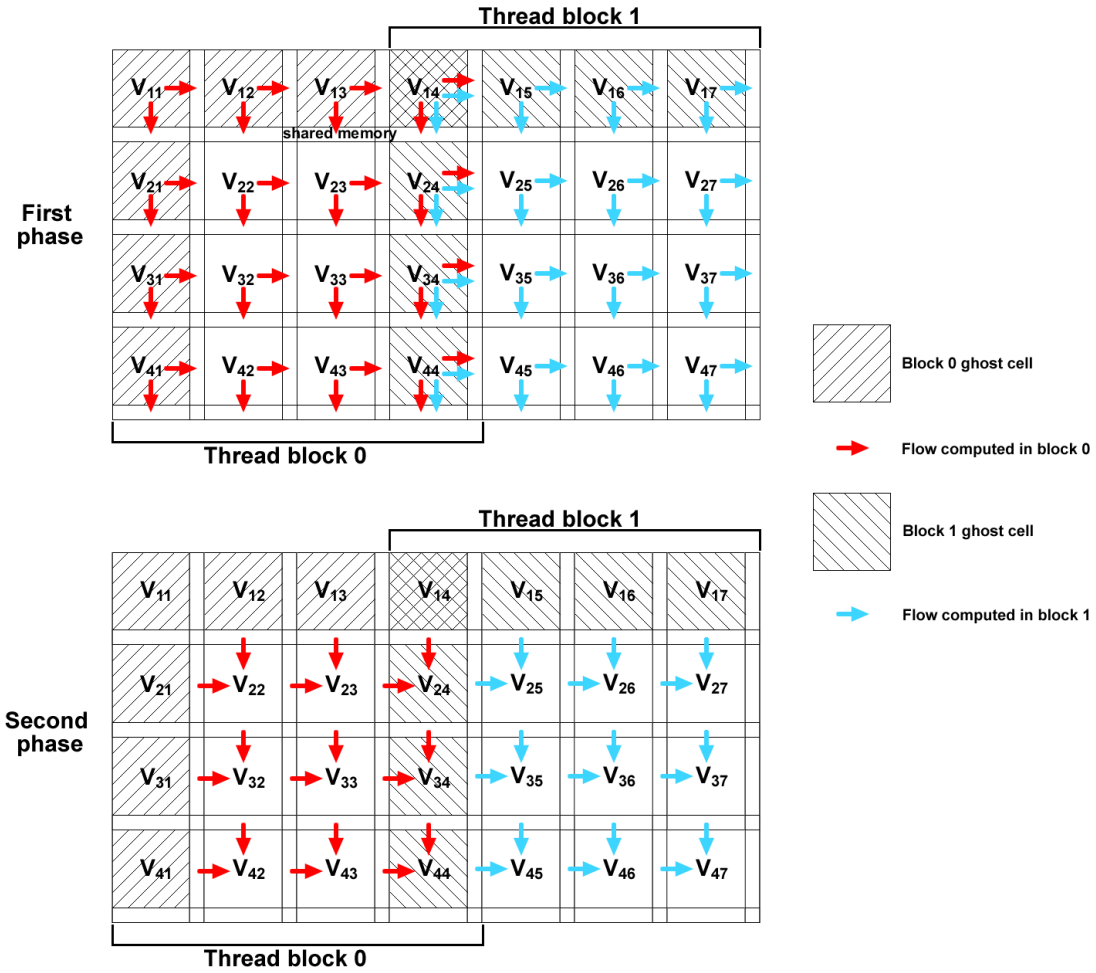


Figure 5. The two phases of the Ghost cell decoupling solution

The first two phases of this approach are illustrated in Figure 5 using a thread block size of 4×4. In the first phase, the flux contributions calculated by the thread block 0 and thread block 1 are depicted. In each volume there are two arrows that simbolize the storage of its right and down flux contributions in buffers allocated in shared memory. The ghost region of a thread block consists of the volumes located in the frontiers of the 4×4 thread block (see shaded boxes in Figure 5). Note that in order to achieve a conflict free concurrent execution of the thread blocks, the computations of some frontier volumes (see volumes V_{14}, V_{24}, V_{34}, and V_{44}) are replicated in both thread blocks. In the second phase, the volumes that do not belong to a ghost region update their flux by accumulating the left and up contributions saved in the shared memory buffers at the end of the first phase. Therefore, only nine (3×3) threads of each block do work in this second phase.

The improvement obtained with the implementation of the ghost cell decoupling technique grows with the thread block size because there are fewer threads that do not work in the second phase. For a block size `blockdimX`×`blockdimY`, the ratio of threads that perform the second stage is given by:

$$\%\, threads = \frac{(blockdimX - 1) \times (blockdimY - 1)}{blockdimX \times blockdimY} \times 100$$

For example, if the block size is 64 (8×8), then 49 (7×7) threads work in the second phase, which represents 76% of the threads in the block. If the block size is 16×16, this ratio increases to 88%.

DOI: 10.1002/cpe

Thus, the percentage of threads by block that do not make the second phase is smaller for larger block sizes.

4.2. *Two-phase reduction*

This improvement consists in executing a part of the reduction of *Stage2* at the end of *Stage1*, thus reducing the amount of work of the reduction kernel of *Stage2*. Following this strategy, in the kernel of *Stage1* each thread block makes a local reduction of the $\Delta t[v]$ calculated by the threads belonging to this block (lines 26-28 of Figure 4) and that have already been stored in shared memory buffers (line 21 of Figure 4).

For example, without this improvement, given a grid of 1000×1000 volumes, there would be 1000000 time steps (one per volume) to be reduced by the reduction kernel. Considering a thread block size of 8×8 in the kernel of *Stage1*, the size of the vector Δt would be $1000000/49 \approx 20408$ elements. Note that the denominator is 49 because although the thread block size is 64 (8×8 threads), only 49 (7×7) of the threads are responsible for computing the time step ($\Delta t[v]$) in *Stage1*. The remaining 15 threads process the volumes of the ghost regions and therefore do not compute time steps. Furthermore, let us realize that the thread block has the 49 values of $\Delta t[v]$ it has computed in shared memory, where the accesses needed to reduce them to a single value are much faster than in global memory. As a result, in this example only 20408 accesses to global memory will be required in the kernel of *Stage2*.

Another important performance consideration is that the thread block size must be well-balanced. On one hand, it must be large enough so that the percentage of threads that perform useful work in the second phase of the *Stage1* is high. On the other hand, it must be small enough to enable the parallel execution of enough thread blocks to keep busy the cores in the device. According to this, we have tried a set of thread block sizes and we have obtained the best performance for 8×8. Note that with size 8×8, each thread block needs 4 KB of shared memory. For a configuration of 48 KB for shared memory, it enables more than 8 simultaneous blocks in a single SM. Finally, this optimization requires another two changes with respect to the naive implementation: a buffer to perform the local reduction in the kernel of *Stage1*, and an adjustment of the grid size of the kernel of *Stage2*.

4.3. *Usage of the texture memory*

Despite the optimization described above, this algorithm still presents uncoalesced accesses to the GPU global memory because of the accesses in the y-direction of the grid of volumes. Specifically, threads that belong to the same *halfwarp* access to different memory segments. Nvidia advises in [14] to use texture memory for these cases, and this exploits its higher bandwidth if there is 2D locality in the texture fetches, avoiding this way uncoalesced loads. Although this recommendation is for devices with compute capability 1.X, in the case of the compute capability 2.X the performance is still better than the one obtained using global accesses and the L1 cache [15], reason why we have applied this optimization.

It is important to mention that we use texture memory both for reads and writes. Nvidia indicates [14] that if the global memory pointed by a texture is overwritten, the texture cache will stay in an inconsistent state and the following reads (within the same kernel) to these texture memory positions will return wrong values. Nevertheless, the arrays that benefit from the texture memory and which are both read and written in our application, never experience both kind of accesses in the same kernel, i.e., they are only either read or written within a given kernel. Thus they can be safely stored in texture memory. The arrays that are accessed by the texture unit are: (1) the array of fluid of the previous iteration, which is stored as a 2D texture of `float4` elements and which changes in each iteration; and (2) the array of parameters, which is stored as a 2D texture of `float` elements and remains constant during the whole simulation. Let us emphasize that these data cannot be stored into constant memory because they require more than 64 KB, which is the maximum of constant memory size for devices of compute capability 2.X.

DOI: 10.1002/cpe

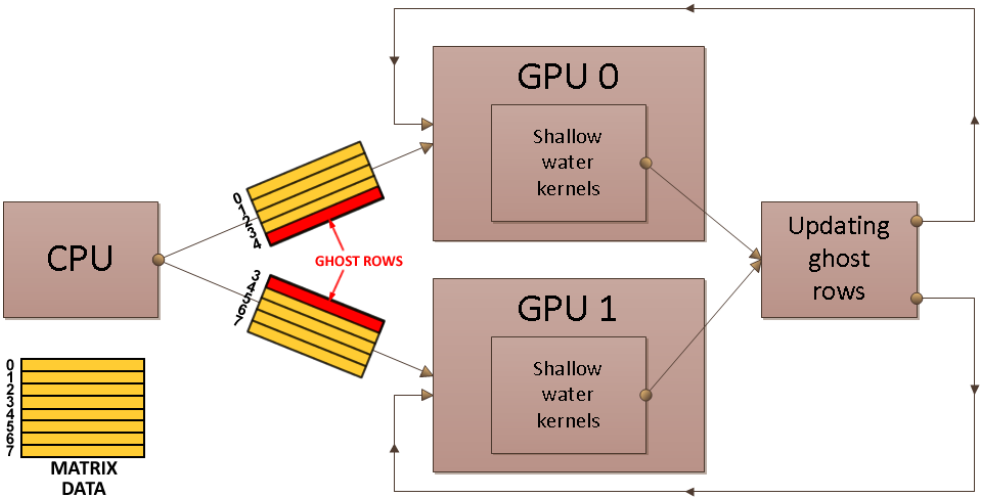


Figure 6. Multi-GPU implementation for two GPUs

In [9] the texture memory is used instead of shared memory meanwhile we store the parameters of the fluid in shared memory but we access to the global memory through the texture memory in order to avoid the uncoalesced accesses when they go in the y-direction of the grid.

5. MULTI-GPU IMPLEMENTATION

The GPU implementations presented in Sections 3 and 4 have been extended to run on multi-GPU systems using MPI [16]. Figure 6 shows the execution flow of our multi-GPU implementations running on two GPUs. Basically, we have an MPI process for each GPU and the workload of the time loop (see lines 2-27 of Figure 2) is distributed among these MPI processes, so that each one of them contains the portion to be processed by its associated GPU. In order to preserve load-balancing, the following data distribution has been done. The key idea is to split the 2D domain by applying a consecutive distribution in the first dimension (also known as row block distribution). The 2D domain is represented by a matrix of 8 rows in the figure. The first GPU is assigned rows 0..3 and the second GPU rows 4..7. In order to compute the flow from all neighbors in *Stage1*, the rows that are in the border of the region assigned to each GPU need the data of neighboring rows assigned to another GPU. Thus, the borderline rows are duplicated in the neighbor GPUs, giving place to read-only ghost regions of one row of size (see ghost row 4 in GPU 0 and ghost row 3 in GPU 1). For the computations to be correct accross iterations of the time loop, the ghost rows must be updated in each time iteration through MPI messages between the processes that own the original row and the ghost row. Overall, the behavior of the MPI parallel program resembles at the MPI process level the behavior of the ghost cell decoupling technique.

The parallel program described above needs one MPI message to update each ghost region in each time step. The number of MPI messages can be reduced by sending more than one ghost row per MPI message. This is, if a process sends `GHOST_ROWS` ghost rows, in the following iteration the receiver process will be able to read these `GHOST_ROWS` ghost rows, and it will update `GHOST_ROWS-1` of them. In the next iteration, the receiver process will read the `GHOST_ROWS-1` ghost rows updated in the previous iteration, which have therefore correct values, and it will update `GHOST_ROWS-2` of them and so on. This way, as long as a process has at least one updated ghost row it can start a new loop iteration without requiring MPI messages to update its ghost rows. Summarizing, when one ghost row is used per border of the region assigned to each GPU, then one MPI message is needed per iteration to maintain the consistency of the ghost region between each two processes. With two ghost rows, the MPI message is only needed every two iterations (although it will be twice larger); with four ghost rows, the MPI message would be only required every four

DOI: 10.1002/cpe

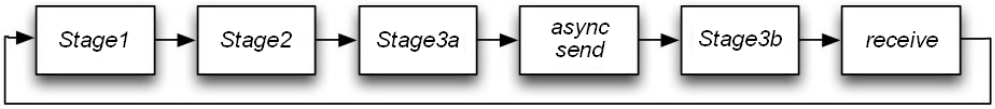


Figure 7. Overlapping of communication and computation in multi-GPU versions

iterations (and it would be four times larger), and so on. This technique is known as Ghost Cell Expansion [17] and it has been used in other works of shallow water simulation [11].

Another important feature of our multi-GPU versions is the use of nonblocking MPI messages in order to overlap communications with GPU computations (see Figure 7). For this, we have splitted the *Stage3* into two separate kernels. The changes in the algorithm with respect to the single-GPU version appear after the global time step reduction of *Stage2*. First, each MPI process updates the rows that are ghost rows at its neighbor processes (*Stage3a*). Then, each process uses nonblocking messages to communicate the ghost rows and, meanwhile, it updates the remaining rows (*Stage3b*). Once this update is done, each process checks the reception of the ghost rows and it updates the GPU memory with these data. At this point, each MPI process can start a new time step iteration.

We also experimented with a version that overlapped the MPI messages with the computations in *Stage1*. First, the flux variations of the ghost rows were calculated. These flux variations were interchanged while those of the remaining rows were calculated. Finally, right before *Stage3*, each MPI process received the flux variations sent from the neighbor processes so that it could update all of its volumes, including those in the ghost rows. The results of this implementation are not discussed because its performance is worse than the version that makes the data interchange during *Stage3*.

Another point where MPI messages are needed is when all the threads finish the kernel of *Stage2*. At this moment each GPU has a local minimum of $\Delta tGlobal$. To complete the reduction process, each process sends its local $\Delta tGlobal$ to a unique process that performs the reduction on CPU.

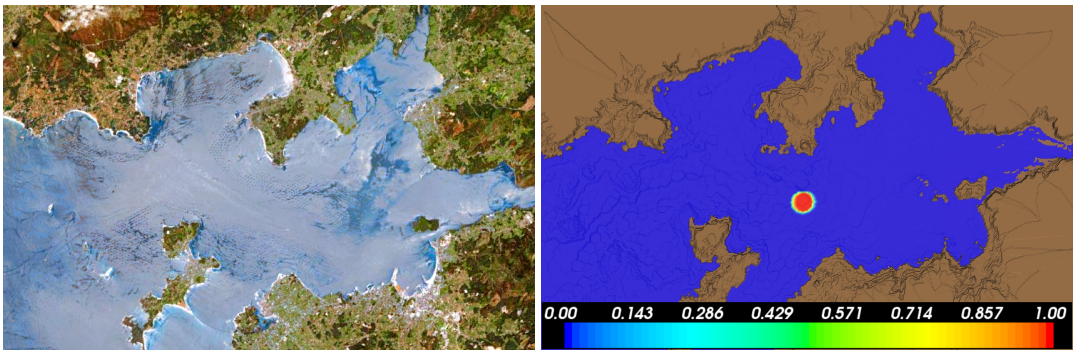
6. EXPERIMENTAL RESULTS

Our evaluation has been performed in a heterogeneus cluster with 2 nodes connected via an Infiniband network. This system is a Nvidia S2050 preconfigured cluster with 4 M2050 GPUs. Each node is directly connected (PCIe) to two M2050 GPUs. Each node has 12 GB of host DDR3 memory and its general purpose CPU is an *Intel Xeon X5650* at 2.67 GHz, with 6 cores and hyperthreading of 2 threads per core reaching a maximum memory bandwidth of 32 GB/s. Each M2050 GPU has 448 *streaming processors* and 3 GB of GDDR5 memory. The software setup is *Debian GNU/Linux 6.0.1 (squeeze)* operating system using *g++ 4.3.5* and *nvcc* 4.0 compilers.

The simulation performed in this work is based on the *Ría de Arousa*, an estuary in Galicia (Spain), whose *GoogleMaps* satellite image is displayed in Figure 8(a). In this test the natural 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. This test makes extensive use of dry-wet fronts in the coastal zones and emerging islands. A discharge of pollutant is artificially added to study its propagation and determine the most affected areas. The total simulated period is 604,800 seconds (one week of real time). Figure 8(b) represents the initial setup where the pollutant is concentrated on a small circle with a radius of 400m. The color scale below indicates the normalized concentration of pollutant. The model has provided an accurate simulation of the disaster evolution (see Figure 8(c)), and thanks to the simulation it was possible to predict the most affected areas. Pollutant discharge not only may have serious environmental consequences, but it can also cause much economical damage to zones where an important part of the local wealth depends on seafood products or tourism.

Table I shows the execution time and the speedups for several mesh sizes. All our implementations use single precision data. The CPU times were taken on the CPU that was described above,

DOI: 10.1002/cpe



(a) Satellite image (GoogleMaps) (b) Initial setup

(c) Pollutant concentration after eight days

Figure 8. Evolution of the Ría de Arousa simulation

Table I. Execution times (in seconds) and speedups

Mesh size	CPU			*single-GPU naive*		*single-GPU optimized*		*multi-GPU naive*		*multi-GPU optimized*	
	sequential time	OpenMP									
		time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
100	932	157	5.92x	12.28	12.78x	11.6	1.06x	16.58	0.74x	16.71	0.69x
200	7440	1086	6.85x	64.96	16.71x	59.04	1.10x	45.10	1.44x	46.49	1.27x
300	23912	3443	6.95x	203.90	16.89x	169.77	1.20x	107.39	1.90x	96.58	1.76x
400	56256	8361	6.73x	447.19	19.71x	387.69	1.15x	187.27	2.38x	172.89	2.24x
500	109201	16527	6.61x	878.55	18.81x	730.09	1.20x	319.10	2.75x	283.41	2.58x
600	188125	28580	6.58x	1455.18	19.64x	1231.97	1.18x	507.43	2.87x	454.75	2.71x
700	297849	45344	6.57x	2343.87	19.35x	1928.40	1.22x	764.76	3.06x	683.94	3.01x
800	443247	67110	6.60x	3322.87	20.20x	2849.49	1.17x	1074.23	3.09x	947.59	3.01x
900	629210	95461	6.59x	4848.76	19.69x	4020.39	1.21x	1549.51	3.13x	1324.24	3.04x
1000	860457	130815	6.58x	6448.80	20.29x	5455.51	1.18x	2035.36	3.17x	1717.92	3.18x

using OpenMP [18] to take advantage of that multicore chip. Since the second thread provided by hyperthreading typically only provides 15% to 20% of the performance of a real core, we can see that our OpenMP implementation is very efficient. The speedups of the naive GPU implementation are calculated with respect to the CPU times. The speedups of the single-GPU optimized version and the multi-GPU naive version have been obtained with respect to the single-GPU naive version. The speedup of the multi-GPU optimized version, has been obtained with respect to the single-GPU optimized one. The processes of the multi-GPU versions share a single ghost row with each neighbor. The smallest mesh takes the minimum time for single-GPU versions. These versions are faster than the multi-GPU versions for this mesh because the ghost rows copies between GPUs offset completely the advantage of the parallelization of the computations on multiple GPUs for this small amount of data. The simulation of the biggest mesh requires about 36 hours in a multithread

DOI: 10.1002/cpe

Table II. Execution times (in seconds) and speedups after applying each improvement separately

Mesh size	Num. Iter.	*single-GPU naive*	*evol-I*		*evol-II*		*single-GPU optimized*	
		time	time	speedup	time	speedup	time	speedup
100 × 100	164243	12.28	11.71	1.05x	11.95	0.98x	11.55	1.03x
200 × 200	335514	64.96	63.35	1.03x	64.84	0.98x	59.04	1.10x
300 × 300	503362	203.90	187.49	1.09x	189.06	0.99x	169.77	1.11x
400 × 400	671293	447.19	424.27	1.05x	426.26	1.00x	387.69	1.10x
500 × 500	839237	878.55	797.99	1.10x	799.51	1.00x	730.09	1.10x
600 × 600	1007255	1455.18	1337.99	1.09x	1334.99	1.00x	1231.97	1.08x
700 × 700	1175349	2343.87	2141.43	1.09x	2139.18	1.00x	1928.40	1.11x
800 × 800	1343453	3322.87	3196.86	1.04x	3128.72	1.02x	2849.49	1.10x
900 × 900	1511582	4848.76	4550.57	1.07x	4458.91	1.02x	4020.39	1.11x
1000 × 1000	1679708	6448.80	6146.36	1.05x	6025.03	1.02x	5455.51	1.10x

CPU implementation and 107 minutes for the single-GPU version based in recomputation. With the optimized single-GPU version presented in this paper, this same simulation takes 91 minutes, and only 29 minutes in the multi-GPU version using 4 GPUs.

6.1. Isolated impact of the improvements applied

Table II shows the evolution of the performance after applying step by step the improvements explained in Section 4 to the naive implementation. It is an incremental development so that each version includes all improvements of the previous ones. The speedups of each version have been measured with respect to the times of the previous version. There are two intermediate versions: *evol-I* and *evol-II*. The *evol-I* version is equal to the *single-GPU naive* version after replacing its first recomputation-based kernel of *Stage1* with the ghost cell decoupling-based kernel (see details in Section 4.1). The *evol-II* version includes additionally the local reduction in the kernel of *Stage1* taking advantage of using shared memory buffers and the subsequent modifications of the size of the kernel of *Stage2* (see Section 4.2). Finally, the last version, *single-GPU optimized*, also contains the last improvement applied in our development, i.e., the usage of texture memory (see Section 4.3).

The *local reduction* improvement evaluated in the *evol-II* column represents a poor contribution to the overall speedup. This improvement is aimed at reducing the work and the number of accesses to global memory of the reduction kernel of *Stage2*. This kernel performs little work for the smaller meshes and applying this improvement has no impact on performance. As the work of the reduction kernel increases, the speedup provided by this optimization grows too. The best improvement percentage is achieved by the usage of the texture memory. The GPU used in this study (Nvidia S2050) is a device with 2.0 compute capability, which has L1 cache for the global memory. The usage of texture memory is more recommended for GPUs of lower compute capabilities because the use of texture cache has a bigger impact in devices that have no L1 cache for their global memory. However, in our case the usage of texture memory means a noticeable 10% of improvement percentage because it optimizes the uncoalesced memory accesses (see details in Section 4.3).

6.2. Impact of communication/computation overlapping

In order to measure the impact on performance of our MPI implementation, we have performed a set of measures of the execution time needed to send/receive the ghost rows, the execution time of the GPU kernel whose time cost we want to hide and the total time of the send/receive operations plus the kernel time. We have used a Gigabit Ethernet network and an Infiniband network. Figure 9 illustrates the study performed including the times with blocking and nonblocking communications. As expected, the communication time is very high for the Gigabit Ethernet network using blocking communications. For this reason using nonblocking communications and communication/computation overlapping halves the total execution time. This improvement is much higher than the one obtained on the Infiniband network. Nevertheless, our overlapping of

DOI: 10.1002/cpe

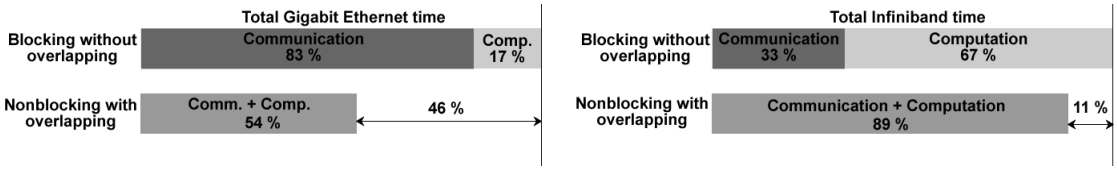


Figure 9. Overlapping communication and GPU computation

Table III. Execution times (in seconds) and speedups with respect to the version using a single ghost row of the multi-GPU optimized version using 2, 4 and 8 ghost rows over an Infiniband network

Mesh size	Num. Iter.	*multi-GPU (2 ghost rows)*		*multi-GPU (4 ghost rows)*		*multi-GPU (8 ghost rows)*	
		time	speedup	time	speedup	time	speedup
100 × 100	164243	12.40	1.35x	10.58	1.58x	10.65	1.57x
200 × 200	335514	39.10	1.19x	35.38	1.31x	32.95	1.41x
300 × 300	503362	84.26	1.15x	79.52	1.21x	75.27	1.28x
400 × 400	671293	156.12	1.11x	148.89	1.16x	147.78	1.17x
500 × 500	839237	270.58	1.05x	258.88	1.09x	261.43	1.08x
600 × 600	1007255	426.85	1.07x	417.31	1.09x	416.25	1.09x
700 × 700	1175349	642.43	1.06x	621.28	1.10x	636.45	1.07x
800 × 800	1343453	902.63	1.05x	909.56	1.04x	903.25	1.05x
900 × 900	1511582	1254.30	1.06x	1239.61	1.07x	1269.25	1.04x
1000 × 1000	1679708	1704.33	1.01x	1680.38	1.02x	1680.73	1.02x

communication and GPU computation still allows us to hide a part of the communication cost and achieve a non negligible 11% reduction of the execution time in this network.

6.3. Impact of the Ghost Cell Expansion technique

In each iteration of the time loop, our multi-GPU version of the shallow water simulator needs to get the current value of the ghost rows used by each process. As this application uses the MPI library, we need to send MPI messages of the size of a row from the process that uses and updates the ghost row, to the process that only uses that row as a ghost row. Thus, one MPI message is needed by time iteration to update each ghost region. In this section, we evaluate the impact of using the multi-GPU version of the optimized implementation the Ghost Cell Expansion strategy described in Section 5. This strategy is based on the use of ghost regions of N rows each, so that the MPI messages required to refresh these regions are N times larger being only needed every N iterations of the time loop.

Table III shows the execution times and the speedups of the multi-GPU optimized version with the Infiniband network, when using ghost regions of 2, 4 and 8 rows. The baseline of the speedups is the version with ghost regions of a single row. The best speedups were obtained for the smaller problems as it was expected. This is because there are fewer computations and we need the same number of messages than for the largest problems, so message passing represents an important part of the total execution time. For larger meshes the speedup obtained is slight. This is due to two reasons: the high performance of the Infiniband network and the ratio between message passing and computation.

The results with an Ethernet network are shown in Table IV. In this case, sharing more than one ghost row between processes has a greater impact on performance. As in the case of the Infiniband network, the best speedups are obtained for the smaller meshes, although in this network the impact is larger. All the times are worse than the times of Infiniband, but this difference is low enough to consider this implementation a very competitive multi-GPU version for both Infiniband and Ethernet. For example, for the largest mesh, the execution time using an Infiniband network is a 15% lower than the execution time of Gigabit Ethernet.

DOI: 10.1002/cpe

Table IV. Execution times (in seconds) and speedups of multi-GPU optimized version using 1, 2, 4 and 8 ghost rows over a Gigabit Ethernet network

Mesh size	Num. Iter.	*multi-GPU (1 ghost row)*	*multi-GPU (2 ghost rows)*		*multi-GPU (4 ghost rows)*		*multi-GPU (8 ghost rows)*	
		time	time	speedup	time	speedup	time	speedup
100×100	164243	78.72	67.08	1.17x	43.00	1.83x	34.93	2.25x
200×200	335514	210.54	121.49	1.73x	100.84	2.09x	81.58	2.58x
300×300	503362	289.54	217.09	1.33x	152.68	1.90x	150.66	1.92x
400×400	671293	405.24	370.88	1.09x	247.34	1.64x	234.65	1.73x
500×500	839237	661.15	559.73	1.18x	403.09	1.64x	346.31	1.91x
600×600	1007255	815.93	747.39	1.09x	562.96	1.45x	592.89	1.38x
700×700	1175349	1085.22	878.17	1.24x	794.83	1.37x	846.37	1.28x
800×800	1343453	1502.48	1281.25	1.17x	1085.69	1.38x	1188.71	1.26x
900×900	1511582	1781.19	1580.05	1.13x	1473.26	1.21x	1622.48	1.10x
1000×1000	1679708	2333.83	2042.44	1.14x	1934.34	1.21x	2022.69	1.15x

Table V. L^1 norm at time T = 1 s for several meshes. The reference solution is CPU sequential

$L^1 error$	100×100	400×400	1000×1000
h	1,10e-7	8,75e-8	1,79e-7
q_x	1,40e-7	1,78e-7	5,79e-7
q_y	9,00e-8	1,28e-7	3,58e-7

6.4. Comparison with a reference CPU implementation

In this section, we measure the accuracy of our GPU simulations with respect to the numerical results of the same test case executed with the CPU sequential version as reference solution. The test used is an academic problem where a water column falls in a water tank so that the generated ripples can be easily tested. Table V shows the value of the L^1 norm for T = 1 second for the meshes 100×100, 400×400, 1000×1000 using the GPU optimized version. The rows of the table show the error for each conformant parameter of the fluid. The measured numerical error for single precision data is negligible, so it does not affect the accuracy of the parallel shallow waters simulator.

7. CONCLUSIONS

In this work we have started from a naive single-GPU implementation for the simulation of pollutant transport in shallow waters. This version was based on a recomputation solution in which redundant computations and many accesses to global memory were performed. An optimized single-GPU version that significantly reduces the number of computations by following a ghost cell decoupling strategy and which exploits shared memory and textures has been implemented. This optimized version achieved an average speedup of 19% with respect to the naive single-GPU implementation for the five largest problem sizes.

We have also developed MPI-CUDA versions of these naive and optimized single-GPU implementations that make efficient usage of multi-GPU systems. Moreover, we have optimized our multi-GPU versions applying ghost cell expansion, which reduces the number of messages by using ghost regions of several rows for the chunks of data assigned to each GPU. The impact on performance of this technique heavily depends on the type of the network connection. This way, while in Infiniband changing the ghost region size from one row to four rows, increases the speedup when using 4 GPUs and the largest mesh from 3.18x to 3.25x (2% of increase) with respect to the single-GPU version, in Gigabit Ethernet the speedup goes from 2.34x to 2.82x (21% of increase). This result, which is very possitive taking into account the penalties of communications, makes this version especially interesting when a high performance network is not available.

For a mesh of 1000×1000 volumes, using 4 GPUs and an Infiniband network and with a ghost region size of four rows, the optimized multi-GPU version simulates the evolution of a realistic environment during seven days in only 28 minutes. Thus, there is a factor of 360 units of real time

DOI: 10.1002/cpe

simulated during a single unit of simulation time. This property is very interesting, as it enables to perform quick studies of the behavior of a pollutant in a realistic environment, under different hypothesis, and fast enough to take all the required decisions to deal with it.

REFERENCES

1. R.J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2002.
2. F. Bouchut. *Nonlinear Stability of Finite Volume Methods for Hyperbolic Conservation Laws and Well-Balanced Schemes for Sources*. Birkhäuser, 2004.
3. M.-O. Bristeau, B. Perthame. Transport of Pollutant in Shallow Water using Kinetic Schemes. *ESAIM: Proc.* 2001; **10**:9–21.
4. Zhengfu Xu, Chi-Wang Shu. Anti-diffusive Finite Difference WENO Methods for Shallow Water with Transport of Pollutant. *J. Comput. Math.* 2006; **24**(3):239–251.
5. E. Audusse, M.-O. Bristeau. Transport of Pollutant in Shallow Water: A Two Time Steps Kinetic Method. *ESAIM: Mathematical Modelling and Numerical Analysis* 2003; **37**(2):389–416.
6. F. Benkhaldoun, I. Elmahi, M. Seaïd. Well-balanced Finite Volume Schemes for Pollutant Transport on Unstructured Meshes. *Journal of Computational Physics* 2007; **226**(1):180–203.
7. 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.* 2008; **221**(1):16–32.
8. A. Brodtkorb, T. Hagen, K. Lie, J. Natvig. Simulation and Visualization of the Saint-Venant System using GPUs. *Computing and Visualization in Science*. 2010; **13**(7):341–353.
9. M. de la Asunción, J.M. Mantas, M.J. Castro. Simulation of One-Layer Shallow Water Systems on Multicore and CUDA Architectures. *The Journal of Supercomputing* 2011; **58**:206–214.
10. M. Acuña, T. Aoki. Real-Time Tsunami Simulation on Multi-node GPU Cluster. *ACM/IEEE conference on Supercomputing* 2009 [poster];
11. M. Sætra, A. Brodtkorb. Shallow Water Simulations on Multiple GPUs. *Applied Parallel and Scientific Computing*. 2012; **7134**:56–66.
12. M. Viñas, J. Lobeiras, B.B. Fraguela, M. Arenaz, M. Amor, and R. Doallo. Simulation of Pollutant Transport in Shallow Water on a CUDA Architecture. *2011 International Conference on High Performance Computing and Simulation (HPCS)*, Istanbul, Turkey, 2011; 664–670.
13. NVidia. *Cuda Toolkit 4.1*. URL `http://developer.nvidia.com/cuda-toolkit-41`, accessed on March 10, 2012.
14. NVidia. *NVIDIA CUDA C Best Practices Guide*. 3.2 edn.
15. J. Lobeiras, M. Amor and R. Doallo. Performance Evaluation of GPU Memory Hierarchy using the FFT. *Proc. of the 11th Interanational Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE)*, vol. 2, 2011; 750–761.
16. W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press: Cambridge, MA, 1999.
17. C. Ding, Y. He. *A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems*. *Proc. of SC2001*, ACM Press, 2001; 50–50.
18. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001.

DOI: 10.1002/cpe