Noname manuscript No.
(will be inserted by the editor)

Simulation of one-layer shallow water systems on multicore and CUDA architectures

Marc de la Asunción · José M. Mantas · Manuel J. Castro

Received: date / Accepted: date

Abstract The numerical solution of shallow water systems is useful for several applications related to geophysical flows but the big dimensions of the domains suggests the use of powerful accelerators to obtain numerical results in reasonable times. This paper addresses how to speed up the numerical solution of a first order well-balanced finite volume scheme for 2D one-layer shallow water systems by using modern Graphics Processing Units (GPUs) supporting the NVIDIA CUDA programming model. An algorithm which exploits the potential data parallelism of this method is presented and implemented using the CUDA model in single and double floating point precision. Numerical experiments show the high efficiency of this CUDA solver in comparison with a CPU parallel implementation of the solver and with respect to a previously existing GPU solver based on a shading language.

Keywords General Purpose computation on Graphics Processing Units (GPGPU) · Shallow water systems · OpenMP · CUDA

1 Introduction

The shallow water equations, formulated under the form of a conservation law with source terms, are widely used to model the flow of a layer of fluid under the influence of gravity forces. The numerical solution of these models is useful for several applications related to geophysical flows, such as the simulation of rivers or dambreaks. These simulations impose great demands on computing

Marc de la Asunción and José M. Mantas
Depto. Lenguajes y Sistemas Informáticos. Universidad de Granada
E-mail: marc@correo.ugr.es, jmmantas@ugr.es

Manuel J. Castro
Depto. Análisis Matemático. Universidad de Málaga
E-mail: castro@anamat.cie.uma.es

power due to the dimensions of the domains (space and time), and very efficient solvers are required to solve these problems in reasonable execution times.

Since the numerical solution of shallow water systems exhibits a lot of exploitable parallelism, several works have dealt with the acceleration of these simulations by using parallel hardware. An interesting numerical scheme to simulate shallow water systems and an efficient parallel implementation of this scheme for a PC cluster are presented in [1]. This parallel implementation has been improved in [2] by using SSE-optimized software modules. Although these improvements have made it possible to obtain results in faster computational times, the simulations still require too much runtime.

Modern Graphics Processing Units (GPUs) offer hundreds of processing units optimized for massively performing floating point operations in parallel and can be a cost-effective way to obtain a substantially higher performance in computationally intensive tasks (see [8] for a review of the topic).

There are previous proposals to port shallow water numerical solvers to GPU platforms In [6], a explicit central-upwind scheme is implemented on a NVIDIA GeForce 7800 GTX card to simulate the one-layer shallow water system and a speedup from 15 to 30 is achieved with respect to a CPU implementation. An efficient implementation of the numerical scheme presented in [1] on GPUs is described in [7], obtaining two orders of magnitude speedup on a NVIDIA Geforce 8800 Ultra card with respect to a monoprocessor implementation. These previous proposals are based on the OpenGL graphics application programming interface [9] and the Cg shading language [3].

Recently, NVIDIA has developed the CUDA programming toolkit [4] consisting in an extension of the C language which facilitates the programming of GPUs for general purpose applications by preventing the programmer to deal with the graphics details of the GPU.

Our goal is to accelerate the numerical solution of shallow water systems by using GPUs supporting CUDA. In particular, the one-layer shallow water numerical solver which is parallelized in [1] and [7] has been adapted to the CUDA architecture to obtain much better response times.

The next section describes the underlying numerical scheme. The sources of parallelism and the CUDA implementation of the numerical solver is described in Section 3. Section 4 presents and analyses the results obtained when the solver is applied to several meshes using several GPUs. Finally, Section 5 summarizes the main conclusions and presents the lines for further work.

2 Numerical Scheme

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

$$\frac{\partial W}{\partial t} + \frac{\partial F_1}{\partial x}(W) + \frac{\partial F_2}{\partial y}(W) = \begin{bmatrix} 0 \\ gh \\ 0 \end{bmatrix} \frac{\partial H}{\partial x} + \begin{bmatrix} 0 \\ 0 \\ gh \end{bmatrix} \frac{\partial H}{\partial y}, \tag{1}$$

with

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

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

To discretize System (1), the computational domain D is divided into L cells or finite volumes $V_i \subset \mathbb{R}^2$, which are assumed to be quadrangles. Given a finite volume V_i, $N_i \in \mathbb{R}^2$ is the centre of V_i, $\aleph_i$ is the set of indexes j such that V_j is a neighbour of V_i; Γ_{ij} is the common edge of two neighbouring cells V_i and V_j, and $|\Gamma_{ij}|$ is its length; $\boldsymbol{\eta}_{ij} = (\eta_{ij,x}, \eta_{ij,y})$ is the unit vector which is normal to the edge Γ_{ij} and points towards the cell V_j [1].

Assume that the approximations at time t^n, W_i^n, have already been calculated. To advance in time, with Δt^n being the time step, the following numerical scheme is applied (see [1] for more details):

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

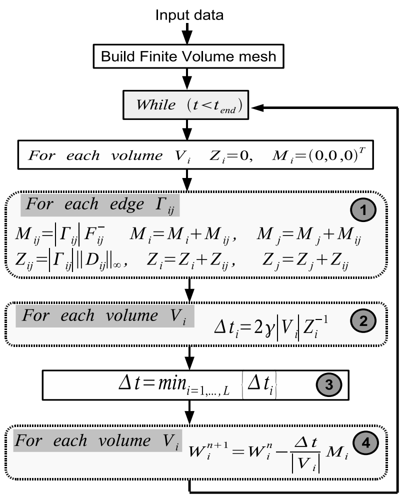
where $|V_i|$ is the area of V_i and $F_{ij}^- \in \mathbb{R}^3$ is a vector whose computation involves many linear algebra operations which depends on W_i^n and W_j^n (see [1]). To compute the nth time step, the following condition can be used:

$$\Delta t^n = \min_{i=1,\dots,L} \left\{ \left[\frac{\sum_{j \in \aleph_i} \mid \Gamma_{ij} \mid \| \mathcal{D}_{ij} \|_\infty}{2\gamma \mid V_i \mid} \right]^{-1} \right\} \tag{3}$$

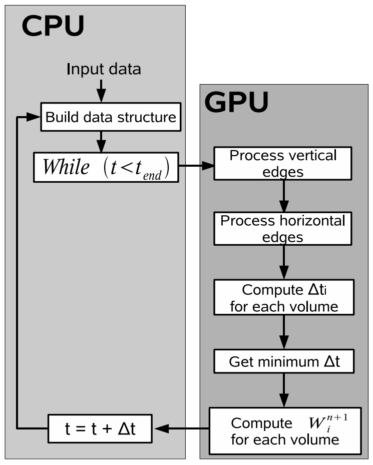
where γ, $0 < \gamma \leq 1$, is the CFL (Courant-Friedrichs-Lewy) parameter and $\mathcal{D}_{ij} \in \mathbb{R}^{3\times 3}$ is a diagonal matrix (see [1]).

3 CUDA Implementation

In this section, we describe the potential data parallelism of the numerical scheme and its implementation in CUDA.


Input data
Build Finite Volume mesh
While $(t<t_{end})$
For each volume V_i $Z_i=0,\ M_i=(0,0,0)^T$
For each edge Γ_{ij}
1
$M_{ij}=|\Gamma_{ij}|F^-_{ij}$ $M_i=M_i+M_{ij}$, $M_j=M_j+M_{ij}$
$Z_{ij}=|\Gamma_{ij}|\|D_{ij}\|_\infty$, $Z_i=Z_i+Z_{ij}$, $Z_j=Z_j+Z_{ij}$
For each volume V_i
$\Delta t_i=2\gamma|V_i|Z_i^{-1}$
2
$\Delta t=min_{i=1,\dots,L}|\Delta t_i|$
3
For each volume V_i
$W_i^{n+1}=W_i^n-\frac{\Delta t}{|V_i|}M_i$
4

(a) Parallelism sources of the numerical scheme



(b) General steps of the parallel algorithm implemented in CUDA

Fig. 1: Parallel algorithm.

3.1 Parallelism sources

Fig. 1a shows a graphical description of the parallelism sources obtained from the mathematical description of the numerical scheme. The main calculation phases are identified with circled numbers, and each of them presents a high degree of parallelism because the computation performed at each edge or volume is independent with respect to that performed at other edges or volumes.

Initially, the finite volume mesh is constructed from the input data. Then the time stepping process is repeated until the final simulation time is reached:

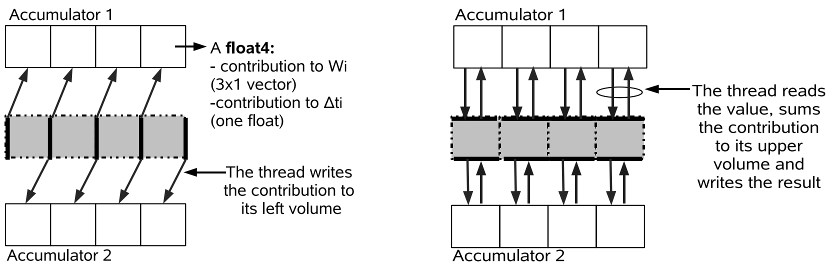
1. **Edge-based calculations**. Two calculations must be performed for each edge Γ_{ij} connecting two cells V_i and V_j $(i,j \in \{1,\dots,L\})$:
   a) Vector $M_{ij} = |\,\Gamma_{ij}\,|\,F^-_{ij} \in \mathbb{R}^3$ must be computed as the contribution of each edge to the calculation of the new states of its adjacent cells V_i and V_j (see Equation (2)). This contribution must be added to the partial sums M_i and M_j associated to V_i and V_j, respectively.
   b) The value $Z_{ij} = |\,\Gamma_{ij}\,|\,\|\,\mathcal{D}_{ij}\,\|_\infty$ must be computed as the contribution of each edge to the calculation of the local Δt values of its adjacent cells V_i and V_j (see Equation (3)). This contribution must be added to the partial sums Z_i and Z_j associated to V_i and V_j, respectively.
2. **Computation of the local Δt_i for each volume**. For each volume V_i, the local Δt_i is obtained as follows (see Equation (3)): $\Delta t_i = 2\gamma\,|V_i|\,Z_i^{-1}$.
3. **Computation of Δt^n**. The minimum of all the local Δt_i values previously computed for each volume V_i is obtained.
4. **Computation of W_i^{n+1}**. The $(n+1)$th state of each volume (W_i^{n+1}) is calculated from the nth state and the data computed in previous phases.

Since the numerical scheme exhibits a high degree of potential data parallelism, it is a good candidate to be implemented on CUDA architectures.

3.2 Implementation details

We consider problems consisting in a bidimensional regular finite volume mesh. The general steps of the algorithm are depicted in Fig. 1b. Each processing step executed on the GPU is assigned to a CUDA kernel. A kernel is a function executed on the GPU, which is executed forming a grid of thread blocks that run logically in parallel (see [4] for more details). Next, we describe each step:

- **Build data structure**. For each volume, we store its state (h, q_x and q_y) and its depth H. We define an array of `float4` elements, where each element represents a volume and contains the former parameters. This array is stored as a 2D texture since each edge (thread) only needs the data of its two adjacent volumes, and texture memory is especially suited for each thread to access its closer environment in texture memory. The per-block shared memory, on the other hand, is more suitable when each thread needs to access many neighbouring elements located in global memory, and each thread of a block loads a small part of these elements into shared memory. We implemented both versions (using a 2D texture and using shared memory) and we got better execution times using a texture.
  The area of the volumes and the length of the vertical and horizontal edges are precalculated and passed to the CUDA kernels that need them.
  We can know at runtime if an edge or volume is a frontier or not and the value of $\boldsymbol{\eta}_{ij}$ of an edge by checking the position of the thread in the grid.
- **Process vertical edges** and **process horizontal edges**. We divide the edge processing into vertical and horizontal edge processing. For vertical edges, $\eta_{ij,y} = 0$, and for horizontal edges, $\eta_{ij,x} = 0$. Therefore, all the operations where these terms take part can be avoided, increasing efficiency. In vertical and horizontal edge processing, each thread represents a vertical and horizontal edge, respectively, and computes the contribution to their adjacent volumes as described in Section 3.1.
  The edges (i.e. threads) synchronize each other when contributing to a particular volume by means of two accumulators stored in global memory, each one being an array of `float4` elements. The size of each accumulator is the number of volumes. Each element of the accumulators stores the edge contributions to the volume (a 3×1 vector M_i and a `float` value Z_i). Then, in the processing of vertical edges, each edge writes the contribution to its right volume in the first accumulator, and the contribution to its left volume in the second accumulator. Next, the processing of horizontal edges is performed in an analogous way with the difference that the contribution is added to the accumulators. Fig. 2 shows this process graphically.
- **Compute Δt_i for each volume**. In this step, each thread represents a volume and computes the local Δt_i of the volume V_i as described in Section



(a) Vertical edge processing (b) Horizontal edge processing

Fig. 2: Computing the sum of the contributions of the edges of each volume.

3.1. The final Z_i value is obtained by summing the two float values stored in the positions corresponding to the volume V_i in both accumulators.

- **Get minimum Δt.** This step finds the minimum of the local Δt_i of the volumes by applying a reduction algorithm on the GPU. The reduction algorithm applied is the kernel 7 (the most optimized one) of the reduction sample included in the CUDA Software Development Kit [4].
- **Compute W_i^{n+1} for each volume.** In this step, each thread represents a volume and updates the state W_i of the volume V_i as described in Section 3.1. The final M_i value is obtained by summing the two 3×1 vectors stored in the positions corresponding to the volume V_i in both accumulators. Since a CUDA kernel cannot write directly into textures, the texture is initially updated by writing the results into a temporary array and then this array is copied to the CUDA array bound to the texture.

We have also implemented a version of this CUDA algorithm using double precision. The differences with respect to the implementation described above are that two arrays of `double2` elements are used for storing the volume data, and four accumulators of `double2` elements are used.

4 Experimental Results

We consider a circular dambreak problem in the $[-5, 5] \times [-5, 5]$ domain. The depth function is $H(x, y) = 1 - 0.4\, e^{-x^2-y^2}$ and the initial condition is:

$$W_i^0(x,y) = \begin{pmatrix} h^0(x,y) \\ 0 \\ 0 \end{pmatrix}, \quad \text{where} \quad h^0(x,y) = \begin{cases} 2 & \text{if } \sqrt{x^2+y^2} > 0.6 \\ 4 & \text{otherwise} \end{cases}$$

The numerical scheme is run for different mesh sizes. Simulation is carried out in the time interval [0,1]. CFL parameter is $\gamma = 0.9$, and wall boundary conditions ($\mathbf{q} \cdot \eta = 0$) are considered.

We also have implemented a serial and a quadcore CPU version (using OpenMP [10]) of the CUDA algorithm. Both versions are implemented in C++, and the Eigen library [5] is used for operating with matrices. We also

Table 1: Execution times in seconds for all the meshes and programs.

Volumes	CPU 1 core	CPU 4 cores	GTX 260 Cg	GTX 260 CUSP	GTX 260 CUDP	GTX 280 Cg	GTX 280 CUSP	GTX 280 CUDP
100×100	0.7	0.2	0.11	0.02	0.07	0.08	0.01	0.06
200×200	5.4	1.5	0.26	0.07	0.37	0.20	0.06	0.36
400×400	44.6	13.9	0.84	0.40	2.75	0.68	0.36	2.63
800×800	358.7	112.3	4.42	2.92	21.45	3.75	2.66	20.81
1600×1600	2883.5	898.1	30.72	23.49	167.5	26.14	21.22	161.4
2000×2000	5639.6	1755.6	58.54	44.97	335.6	49.48	39.83	307.3
2700×2700	13902.4	4340.0	–	111.9	819.7	–	96.63	755.2
3200×3200	23290.0	7240.0	–	184.9	–	–	163.4	–

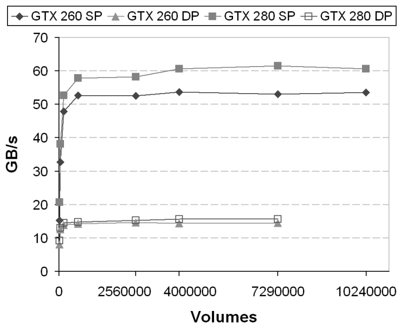
have compared the CUDA implementations with the Cg program described in [7]. We have used the `double` data type in CPU.

All the programs were executed on a Core i7 920 with 4 GB RAM. Graphics cards used were a GeForce GTX 260 and a GeForce GTX 280. Table 1 shows the execution times in seconds for all the meshes and programs (some cases could not be executed due to insufficient memory errors). As can be seen, the number of volumes and the execution times scale with a different factor because the number of time steps required for the same time interval also augments when the number of cells is increased (see Equation (3)). The execution times of the single precision CUDA program (CUSP) outperform that of Cg in all cases with both graphics cards. Using a GTX 280, CUSP achieves a speedup of more than 140 with respect to the monocore version. The double precision CUDA program (CUDP) has been about 7 times slower than CUSP for big problems in both graphics cards. As expected, the OpenMP version only reaches a speedup less than four with respect to the monocore program. Fig. 3 shows graphically the GB/s and GFLOPS obtained in the CUDA implementations with both graphics cards. In the GTX 280 card, CUSP achieves 61 GB/s and 123 GFLOPS for big meshes. Theoretical maximums are: for the GTX 280, 141.7 GB/s, and 933.1 GFLOPS in single precision, or 77.8 GFLOPS in double precision; for the GTX 260, 111.9 GB/s, and 804.8 GFLOPS in single precision, or 67.1 GFLOPS in double precision.

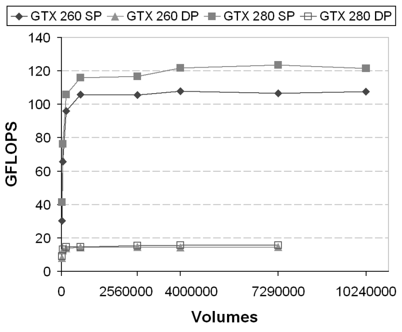
We also have compared the numerical solutions obtained in the monocore and the CUDA programs. The L1 norm of the difference between the solutions obtained in CPU and GPU at time $t = 1.0$ for all meshes was calculated. The order of magnitude of the L1 norm using CUSP vary between 10^{-2} and 10^{-4}, while that of obtained using CUDP vary between 10^{-13} and 10^{-14}, which reflects the different accuracy of the numerical solutions computed on the GPU using single and double precision.

5 Conclusions and further work

An efficient first order well-balanced finite volume solver for one-layer shallow water systems has been derived and implemented using the CUDA framework. This solver implements optimization techniques to parallelize efficiently



(a) GB/s



(b) GFLOPS

Fig. 3: GB/s and GFLOPS obtained with the CUDA implementations in all meshes with both graphics cards.

the numerical scheme on the CUDA architecture. Simulations carried out on a GeForce GTX 280 card using single precision reached 61 GB/s and 123 GFLOPS, and were found to be up to two orders of magnitude faster than a monocore version of the solver for big-size uniform problems, and also faster than a GPU version based on a graphics-specific language. These simulations also show that the numerical solutions obtained with the solver are accurate enough for practical applications, obtaining better accuracy using double precision than using single precision. As further work, we propose to extend the strategy to enable efficient simulations on irregular meshes and to address the simulation of two-layer shallow water systems.

Acknowledgements J. Mantas acknowledges partial support from the DGI-MEC project TIN2007-29664-E and the DGI-MEC project MTM2008-06349-C03-03. M. Castro acknowledges partial support from DGI-MEC project MTM2006-08075.

References

1. Castro MJ, García-Rodríguez JA, González-Vida JM, Parés C (2006) A parallel 2d finite volume scheme for solving systems of balance laws with nonconservative products: Application to shallow flows. Comput Meth Appl Mech Eng 195:2788-2815.
2. Castro MJ, García-Rodríguez JA, González-Vida JM, Parés C (2008) Solving shallow-water systems in 2D domains using Finite Volume methods and multimedia SSE instructions. J Comput Appl Math 221:16-32.
3. Fernando R, Kilgard MJ (2003) The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley.
4. NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home.html. Accessed November 2009.
5. Eigen 2.0.9. http://eigen.tuxfamily.org. Accessed November 2009.
6. Hagen TR, Hjelmervik JM, Lie K-A, Natvig JR, Henriksen MO (2005) Visual simulation of shallow-water waves. Simul Model Pract Theory 13:716-726.
7. Lastra M, Mantas JM, Ureña C, Castro MJ, García JA (2009) Simulation of Shallow-Water systems using Graphics Processing Units. Math Comput Simul 80:598-618.
8. Rumpf M, Strzodka R (2006) Graphics Processor Units: New Prospects for Parallel Computing. Lecture Notes in Computational Science and Engineering 51:89-132.

9. Shreiner D, Woo M, Neider J, Davis T (2007) OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1, Addison-Wesley Professional.
10. Chapman B, Jost G, van der Pas R (2007) Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press.