Parallel GPU implementations of numerical methods for fluid dynamics

Pablo Ezzatti[1] and Sergio Nesmachnow[1]

Centro de Cálculo, Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay, {pezzatti,sergion}@fing.edu.uy

Abstract. This article presents the application of parallel computing techniques using Graphic Processing Unit (GPU) in order to improve the computational efficiency of numerical methods applied to fluid dynamics problems. In the last ten years, GPUs have emerged as a major paradigm for solving complex problems using parallel computing techniques. Fluid dynamics problems usually requires large execution times to perform simulations for realistic scenarios. In this work, two numerical models for fluid dynamics are presented, and parallel implementations on GPU for the Strongly Implicit Procedure and the Cyclic Reduction methods for solving linear systems are introduced. The experimental evaluation of the proposed methods demonstrates that a significant reduction on the computing times can be attained when solving linear systems with representative dimensions, and preliminary results show that the efficiency gains also propagate to the numerical models for fluid dynamics.

Keywords: GPU computing, linear system solvers, fluid dynamics models.

1 Introduction

Numerical models have been widely used for the simulation of water flows and tides in the last forty years. The research at Universidad de la República has been focused in efficient implementations of numerical methods for computational fluid dynamics using high performance computing techniques, mainly for studying the hydrodynamic of the Río de la Plata.

Two of the main numerical models used for studying the Río de la Plata are `MOHID` [7] and `caffa3d.MB` [17]. These are powerful models that allow to simulate many features and hydrodynamic properties. However, the main drawback of these models is the large amount of time required to perform simulations for realistic scenarios. Traditional high performance computing techniques, such as parallel domain decomposition in clusters, have been applied to improve the efficiency of these numerical models. A second line of work, which often requires less economic investment, consist in applying high performance computing techniques using Graphic Processor Units (GPU) to perform the computations. The GPU hardware has shown an excellent relationship between cost and computing power, achieving more FLOPS per dollar than the traditional parallel architectures in many application areas [26].

This article presents the analysis of parallel numerical algorithms developed in GPU, applied to the `MOHID` and `caffa3d.MB` models for computational fluid dynamics. An analysis of the computational performance of the two models is performed just before introducing the parallel versions proposed for speeding-up the computation using GPUs. The preliminary results demonstrate that large values of the acceleration factor (up to 8.7×) are obtained when using a GPU card that costs less than 30% of a standard multicore computer, such as the one used in the computational efficiency analysis.

The content of the manuscript is structured as follows. Next section presents the studied numerical models for computational fluid dynamics. A brief introduction to GPU computing and the review of the related work is presented in Section 3. Section 4 describes the GPU implementations of two methods for solving linear systems. The experimental evaluation of the proposed methods is reported in Section 5, where the efficiency results are also analyzed. Last, Section 6 formulates the conclusions of the research and the main lines for future work.

2 Numerical models for computational fluid dynamics

This section introduces the main features of the `MOHID` and `caffa3d.MB` models, and it also highlights the importance of the methods for solving linear systems of equations in order to achieve efficient simulations.

2.1 The `caffa3d.MB` model

The `caffa3d.MB` model implements the finite element method, applied to the 3D numerical simulation of viscous and/or turbulent fluids with generic scalars transport. The domain geometry is represented by block-structured curvilinear grids, which allow to describe complex scenarios. The interface between blocks is fully implicit, in order to avoid downgrading both the performance and the numerical results of the method. The model allows incorporating rigid objects in the domain to study their interactions with the fluid.

The mathematical model used in `caffa3d.MB` considers the Navier-Stokes equations for mass balance (1) and momentum balance (2) for a non-compressible Newtonian flow, and the conservation equation for a generic passive scalar ϕ (often the temperature) (3).

$$\int_S (\boldsymbol{v} \cdot \hat{n}_S)\, dS = 0 \tag{1}$$

$$\begin{aligned} \int_\Omega \rho \frac{\partial u}{\partial t}\, \partial\Omega + \int_S \rho u (\boldsymbol{v} \cdot \hat{n}_S)\, dS = \int_\Omega \rho\beta\,(T - T_{ref})\,\boldsymbol{g}\,\partial\Omega + \\ \int_S (-p\hat{n}_S)\, dS + \int_S (2\mu D \cdot \hat{n}_S)\, dS \end{aligned} \tag{2}$$

$$\int_\Omega \rho \frac{\partial \phi}{\partial t}\, \partial\Omega + \int_S \rho\phi (\boldsymbol{v} \cdot \hat{n}_S)\, dS = \int_S \Gamma\,(\nabla\phi\,\hat{n}_S)\, dS \tag{3}$$

The caffa3d.MB model was implemented using the original 2D caffa model by Ferziger and Peric [17] as a baseline. The source code of caffa3d.MB is publicly available at the website www.fing.edu.uy/imfia/caffa3d.MB.

Solving the equations in the caffa3d.MB model implies the resolution of band linear systems: pentadiagonal matrices are used in 2D scenarios and heptadiagonal matrices are used in 3D scenarios, regarding the neighboring points in the discretization. The Strongly Implicit Procedure (SIP) solver by Stone [30] is used for solving the linear systems. The SIP solver is an iterative method that splits the linear system resolution in two stages. The first stage calculates an incomplete LU factorization and computes an initial solution. The second stage refines the previous result taking into account the residual and the matrices $\tilde{L}$ and $\tilde{U}$ previously computed in the factorization.

2.2 The MOHID model

The MOHID model implements a finite volume discretization to solve the 3D Navier-Stokes equations using a direct hydrostatic approximation of the pressure fields. This method has been used to study the hydrodinamics and the sediments in the Río de la Plata [18], by considering many variables such as velocity, elevation of the free surface, water salinity, sediment concentration, etc.

By using nested grids with increasing resolution, the MOHID model allows studying specific areas in the domain, obtaining high precision in the results by including as boundary conditions the results obtained using a higher level grid.

In order to perform large-scale simulations over the Río de la Plata and South Atlantic Ocean, MOHID is able to include accurate atmospheric models to propagate the meteorological conditions. A large amount of computing power is needed to perform these simulations, mainly due to the computations required to solve the huge linear systems involved.

MOHID uses the Alternating Direction Implicit (ADI) method to solve the partial difference equations, by dividing the time step into several substeps and solving the equations for each space direction. The ADI method requires to solve tridiagonal linear systems. In the MOHID model, this task is performed using the tridiagonal matrix algorithm, also known as the Thomas algorithm [11], a special case of the Gaussian elimination method applied to tridiagonal systems.

2.3 Efficiency of the linear system resolution

The time required to perform the linear system resolution strongly impacts in the computational efficiency of both the caffa3d.MB and the MOHID models. In the caffa3d.MB model, 40% of the total execution time is required to perform the SIP method in the linear system resolution [31], while in the MOHID model, 50% of the total execution time is required to solve the linear systems using the Thomas algorithm [5]. This fact suggests that improved versions of the models can be developed by including high performance computing techniques into the linear system resolution. These improved models would help to tackle large problem cases and solving more complex simulations in realistic large-scale scenarios.

3 GPU computing

The GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate in the remaining computations. Nowadays, the GPUs have a considerably large computing power, provided by dozens or even hundreds of processing units with reasonable fast clock frequencies. So, in the last ten years, GPUs have been used as a powerful intrinsically parallel hardware architecture to achieve efficiency in the execution of applications.

This section presents the main concepts about GPU programming and a summary of the related works that have proposed applying GPU-based algorithms to speed-up the efficiency of numerical methods.

3.1 GPU programming

Ten years ago, when GPUs were first used to perform general-purpose computation, they were programmed using low-level mechanism such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX [16]. Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA [25], a software architecture for managing the GPU as a parallel computing device without requiring to map the data and the computation into a graphic API.

CUDA is based in an extension of the C language, and it is available for graphic cards GeForce 8 Series and superior, using the 32 and 64 bits versions of the Linux and Windows (XP and successors) operating systems. Three software layers are used in CUDA to communicate with the GPU (see Figure 1): a low-level hardware driver that performs the data communications between the CPU and the GPU, a high-level API, and a set of libraries that includes CUBLAS for linear algebra calculations and CUFFT for Fourier transforms calculation.

For the CUDA programmer, the GPU is a computing device which is able to execute a large number of threads in parallel. A specific procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program –named *kernel*– is loaded in the GPU. The GPU has its own DRAM, and the data are copied from the DRAM of the GPU to the RAM of the host (and viceversa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one of them having eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor of the graphic card, and the blocks are grouped in *grids*. Each time that a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor.

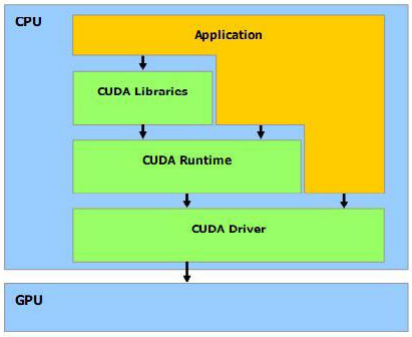

Fig. 1. CUDA architecture.

When a multiprocessor receives one (or more) blocks to execute, it splits the threads in *warps* –a set of 32 consecutive threads–. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Otherwise, the warp serializes the threads. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads are able to access the data using three memory spaces: the *shared memory* of the block, which can be used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memory spaces (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve efficiency in GPU programming. On the other side, the shared memory is placed within the GPU chip, thus it provides a faster way to store the data.

3.2 Related work: numerical methods in GPU

In the first years of the 2000s decade, the pioneering works on applying GPU programming to numerical models proposed GPU implementations of the traditional algorithms for linear system resolution. Rumpf and Strzodka [27] solved linear systems related to finite element models using 8-bits numbers and a GPU implementation of the Jacobi method. Bolz et al. [6] presented GPU implementations of a multigrid method and the Conjugate Gradient (CG) method for sparse and full matrices, respectively. The CG implementation was based in the multiplication of sparse matrices and vectors in the GPU. Another implementation of the multigrid method in GPU was presented by Goodnight et al. [22] to solve boundary problems. Bajaj et al. [2] applied GPUs for solving differential equations using the fourth-order Runge-Kutta method and the Gauss-Seidel solver. A specific Poisson equation derived from the Navier-Stokes equations was solved using a Jacobi method on a grid with 256000 points. GPU-based implementations emerged from these first proposals as promising methods for improving the efficiency of numerical algorithms.

In a second stage, several researchers developed its own numerical applications using specific tools for GPU programming, often providing low portability to other systems. Sorensen and Mosegaard [29] focused on finite element simulations in GPU, addressing the resolution of the stiffness linear systems using CG for full, band, and non-structured sparse matrices. The first proposal of a GPU-based implementation for direct methods applied to sparse linear systems was presented by Christen et al. [10], who evaluated different options to extend the PARDISO library for parallel matrix algorithms [28] using the GPU. Buatois et al. [8] implemented the GC method with a Jacobi preconditioner for sparse matrices by using a block compressing strategy and a reordering technique.

After the release of CUDA, several works have tackled numerical applications following a high-level approach. Göddeke et al. [19, 21] developed GPU implementations of the multigrid and the Gauss-Jordan methods for solving finite element problems, and studied the impact of using single precision and native and emulated double precision arithmetics. Later [20], the authors presented a two-level parallel model combining domain decomposition in a cluster and the GPU-multigrid within each processor, applied to a generic framework for solving finite element problems. Baboulin et al. [1] implemented the LU and Choleski factorizations and the iterative FGMRES-GMRES method in a set or Cell processors, and studied the iterative refinement techniques to improve the results. Barrachina et al. [4] compared the performance of CUBLAS –the implementation of the BLAS library from NVIDIA– with the GotoBLAS implementation, for solving linear systems using padding techniques and mixed strategies for CPU-GPU calculation. Several methods were considered in the comparison, as well as working with single and double precision and using iterative refinement techniques. The framework for solving linear system by Feng and Li [15] used a multigrid method and an hybrid GPU-CPU computation strategy applied to a power grid problem. The authors analyzed the "speedup" for the parallel GPU implementation (i.e. the acceleration factor in the computing times between an execution in CPU and an execution in GPU), and they reported acceleration values up to 15× when compared with a CPU multigrid solver.

Following a different line of work, Michalakes and Vachharajani [23] analyzed the transformation of a fluid numerical model for atmospheric simulations to execute in GPUs, by using the automatic code translator tools from FORTRAN to CUDA. Promising results were achieved for realistic scenarios. Recently, Zhang et al. [32] studied strategies for the efficient resolution of tridiagonal systems using GPUs. The cyclic reduction and the recursive doubling methods were evaluated, obtaining "speedup" values up to 12.5× for small-sized matrices.

Summarizing, many works have recently applied GPUs in order to improve the computational efficiency of numerical methods for linear system resolution. However, there have been few works proposing generic implementations that can be used as black boxes inside complex numerical models. So, there is still room to contribute in this line of research, by proposing efficient implementations of numerical methods applied to generic models in fluid dynamics, such as the ones studied in this work.

4 Speeding-up the efficiency of MOHID and caffa3d.MB using GPUs

This section introduces the proposals oriented to speed-up the computational efficiency of the studied models for fluid dynamics by using GPU computing.

It was already commented in Section 2 that the crucial step to diminish the execution time of the studied models is the linear system resolution. So, the methodology used in this work involves:

1. to isolate the linear system resolution stage in the numerical model.
2. to develop and/or to adapt numerical methods that allow to take advantage of the GPU hardware available.
3. to evaluate the proposed parallel methods.
4. to incorporate the improved methods to the numerical models.

Our group of work has successfully applied the previously described methodology to other numerical models for fluid dynamics, such as the RMA-10 [14] and the RMA-10 [13]. For the RMA-10, efficiency improvements up to 4.14× were obtained in the linear system resolution, and the impact in the efficiency of the whole model was 3.50×. On the other side, efficiency improvements up to 2.76× were obtained in the linear system resolution of the RMA-11 model, and the impact in the efficiency of the whole model was 2.22×. In addition, a shared memory parallel approach applied to caffa3d.MB was able to obtain speedup values up to 1.6× when executing in multiprocessors.

The previous results demonstrate that significant improvements in the computational efficiency of the numerical models for fluid dynamics can be achieved when using efficient solvers for the linear system resolution. Next sections describe the specific proposals to integrate GPU computing techniques to the MOHID and caffa3d.MB models.

4.1 GPU-parallel SIP in caffa3d.MB

The caffa3d.MB uses the SIP solver to solve pentadiagonal and heptadiagonal linear systems. No proposals for a GPU-parallel SIP solver were found in the review of the related work, so a parallel SIP implementation was developed following the methodology previously used in the parallel versions on traditional shared memory hardware. In addition, some ideas from the work by Deserno et al. [12] were adapted to use the GPU as the execution platform for the 2D version of the SIP solver.

The algorithm receives the five diagonals (A_w, A_s, A_p, A_n y A_e, following the geographic nomenclature by Ferziger and Peric [17]), the parameter α, the independent term, and the number of refining iterations. Analyzing the data dependencies, in order to process a certain point (i, j) of the grid, the values of $(i, j-1)$ and $(i-1, j)$ are needed. So, the processing can be concurrently performed by grouping the points of the grid in diagonals (see Figure 2).

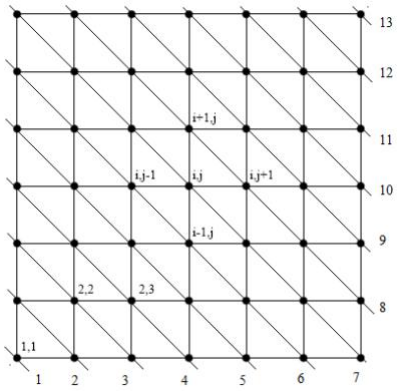

Fig. 2. Processing in the GPU-parallel SIP solver.

The implementation of the SIP solver in the GPU uses the following strategy:

- the method iterates for each diagonal, calling the kernel of the function that performs the parallel computation (including the routines to compute the LU factorization, to compute the residual, and to perform the forward and backward substitutions).
- for each diagonal, the method calculates the threads and the number of blocks needed, given a fixed block size.
- the point (i, j), associated to the thread identification, is computed inside each kernel function,

4.2 Thomas and GPU-parallel CR in `MOHID`

The `MOHID` model uses the Thomas method for solving tridiagonal linear systems. Since the Thomas method is intrinsecally sequential due to the data dependencies, a parallel version of the Cyclic Reduction (CR) method was implemented to improve the `MOHID` efficiency. The CR algorithm performs more arithmetic operations than the Thomas method, but the parallel structure of the data flow makes the method useful for achieve efficiency gains in parallel computers.

Two phases are clearly identified in the CR method: the forward substitution or *elimination* phase and the backward substitution. The elimination phase performs a reduction process that recombines the equations in triplets, grouping the unknowns and eliminating the equations with odd indexes, so the transformed system has half of the number of equations in the original system. This phase is applied until having a single equation. The second phase solves this unique equation and applies the backward substitution for solving the original system.

The elimination phase executes $\log n$ times, applying the forward substitution process in each step. In the GPU implementation, one kernel is used for each equation. The backward substitution phase computes in parallel the unknowns with even indexes, also using one kernel for each equation.

5 Experimental analysis

This section presents the experimental evaluation of the proposed GPU implementations for the linear system solvers studied.

5.1 Development and execution platform

The algorithms were developed using CUDA. The experimental evaluation was performed in a Pentium Dual Core E5200 with a NVIDIA 9800 GTX+ graphic card, whose details are presented in Table 1. The matrices used in the experimental evaluation were obtained from representative scenarios for each numerical model. In order to avoid possible distortions on the execution times due to the non-deterministic nature of the parallel programs, the tables in this section report the average execution times obtained in five executions of the algorithms for each problem instance.

processor/ graphic card	**# cores**	**clock frequency** (MHz)	**L2 cache** (MB)	**memory** (MB)
Pentium DualCore E5200	2	2500	2	2048
Nvidia 9800 GTX+	128	738	-	512

Table 1. Details of the computing platform.

5.2 SIP solver in the `caffa3d.MB` model

Table 2 presents the execution times (in seconds) for the CPU and the GPU implementations of the SIP solver and the acceleration factor obtained when using the new parallel implementation, for representative matrices with several dimensions. The acceleration factor values are simply computed as the quotient between the execution times of the CPU-version and the GPU-version of the SIP solver. Figure 3 summarizes the acceleration factor variation for each problem dimension faced.

matrix dimension	**SIP in CPU** (s)	**SIP in GPU** (s)	**acceleration factor**
1156×1156	0.01	0.11	0.10
16900×16900	0.62	0.51	1.21
264196×264196	3.70	3.18	1.16
1052676×1052676	9.99	4.54	2.20

Table 2. Execution times: SIP in CPU and in GPU.

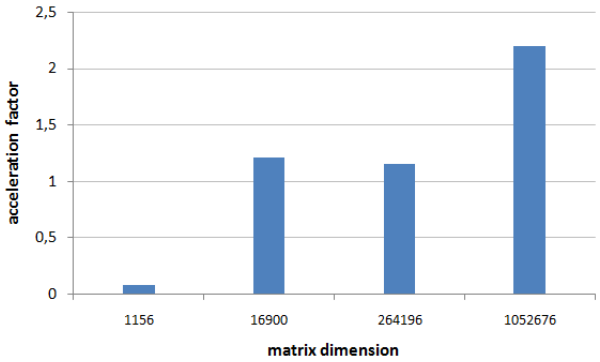

Fig. 3. Speedup of the GPU implementation of the SIP solver.

The results in Table 2 and Figure 3 demonstrate that significant acceleration factor values can be achieved for the largest problem instances using the GPU implementation of the SIP solver. As a reference baseline, the application of shared memory techniques in `caffa3d.MB` obtained a maximum acceleration factor value of 1.6 using two processors [31]. The acceleration factor values reported in Table 2 should increase to larger values when addressing the 3D version of the `caffa3d.MB` model.

5.3 Thomas and CR in the `MOHID` model

Regarding the `MOHID` model, the new parallel CR method was compared against the best solver among the ones implemented in CPU (i.e., the Thomas method). Table 3 presents the execution times for the CPU implementation of the Thomas method and the GPU implementation of the CR solver (in seconds), and the acceleration factor obtained when using the new parallel implementation, for representative matrices with several dimensions. Figure 4 summarizes the acceleration factor variation for each problem dimension faced.

matrix dimension	**Thomas in CPU** (s)	**CR in GPU** (s)	**acceleration factor**
65536×65536	0.008	0.003	2.66
4194304×4194304	0.229	0.066	3.47
16777216×16777216	0.919	0.212	4.34
33554432×33554432	1.832	0.211	8.68

Table 3. Execution times: Thomas in CPU and CR in GPU.

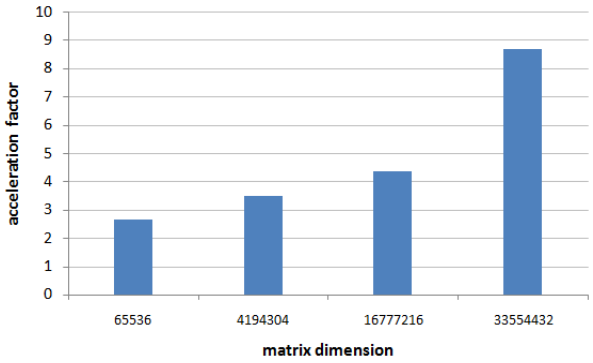

Fig. 4. Speedup of the GPU implementation of the CR solver.

The results in Table 3 demonstrate that large acceleration factor values can be obtained when using the parallel GPU implementation of the CR method. The acceleration factor values linearly increase, achieving up to 8.7× for the largest matrix dimension studied, such as the ones used to model realistic scenarios in the Río de la Plata.

In addition, very large problem instances were also studied, and the results confirmed the advantage of using the parallel GPU implementation of the CR method for solving large scenarios (obtaining acceleration factor values up to 38×). However, the analysis demonstrated that the computational efficiency of the sequential Thomas algorithm in CPU are largely affected by hardware limitations, such as the available amount of RAM.

5.4 Summary and impact in the numerical models

The experimental evaluation has shown that significant improvements on the computational efficiency are attainable when using the GPU-based implementations of the studied linear system solvers. For the SIP solver, a sub-linear growing behavior of the acceleration factor was detected, while linear increasings were obtained when using the CR method. These results are somehow consistent with the complexity order of each method: the SIP solver has linear complexity, so a linear growing behavior in the acceleration factor of the GPU-parallel implementation is expected at most. On the other side, the CR method has logarithmic complexity while the Thomas method has linear complexity, so a linear growing behavior of the acceleration factor values is expected at least.

Preliminary experiments, which have been not formalized yet, have shown that the efficiency gains obtained in the linear system resolution propagate to the numerical models for fluid dynamics. These results, and the previous experiences with other numerical models suggest that including the GPU implementations of the proposed solvers allows to design more powerful and efficient versions of the `MOHID` and the `caffa3d.MB` models, able to tackle large scenarios and perform large simulations in reduced execution times.

Improved efficiency results are expected when developing parallel solvers using a more powerful GPU architecture, such as the new Tesla server recently available in our research group.

6 Conclusions and future work

This work has presented the initial studies on applying GPU computing in order to speed up the execution of numerical models for fluid dynamics. Two methods for solving band linear systems were implemented in GPU using CUDA. The implementations of the SIP and the CR methods were evaluated using matrices obtained from representative scenarios for each numerical method. The experimental analysis demonstrated that the GPU implementations significantly improves over the execution times of the traditional CPU implementations. The GPU implementation of the SIP solver obtained acceleration factor values up to 2.2× when compared with the CPU implementation. The CR method attained significantly large acceleration factor values, up to 8.7× for the largest problem faced. Both methods were able to improve their computational efficiency values when solving the largest problem instances, showing a good scalability behavior. A preliminary analysis suggested that the efficiency improvements in the linear system resolution propagate to the numerical model for fluid dynamics.

The main lines for current and future work focus on implementing other methods for the linear system resolution as well as integrating the GPU implementations with the numerical models. Regarding the first issue, a three-dimensional version of the SIP solver is currently been implemented on GPU, in order to fully exploit the capabilities of the `caffa3d.MB` model. Several alternatives to the CR algorithm, such as the Parallel Cyclic Reduction and Recursive Doubling methods, should also be implemented on GPU, to provide even more efficient strategies for the linear system resolution. In addition, the development of hybrid methods that combine GPU and CPU computation should be addressed in order to increase the efficiency of the proposed methods, possibly by using modern infrastructures that incorporate several GPUs able to work in parallel. Regarding the second line for future work, a more general mechanism for integrating the proposed GPU implementations to the numerical models has to be developed, by designing an independent set of modules that make possible the extension of the proposed approach to other numerical models.

References

1. M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. MIMS EPrint 2009.2, Manchester Institute for Mathematical Sciences, University of Manchester, UK, 2009.
2. Ch. Bajaj, I. Ihm, J. Min, and J. Oh. SIMD optimization of linear expressions for programmable graphics hardware. *Computer Graphics Forum*, 23(4):697–714, 2004.

3. S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, pages 739–748, Berlin, Heidelberg, 2008. Springer-Verlag.
4. S. Barrachina, M. Castillo, F. Igual, R. Mayo, E. Quintana-Ortí, and G. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. Technical report, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Castellón, España, 2008.
5. I. Barreto, P. Ezzatti, and M. Fossati. Estudio inicial del modelo MOHID. Technical report, Universidad de la República, Uruguay, 2009.
6. J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transaction on Graphics*, 22(3):917–924, 2003.
7. F. Braunschweig, P. Leitao, L. Fernandes, P. Pina, and R. Neves. The object oriented design of the integrated water modelling system mohid. In *Computational Methods in Water Resources*, 2004.
8. L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: An efficient sparse linear solver o n the GPU. In *High Performance Computation Conference*, volume 4782 of *Lecture Notes in Computer Science*. Springer, 2007.
9. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transaction on Graphics*, 23:777–786, 2004.
10. M. Christen, O. Schenk, and H. Burkhart. General-purpose sparse matrix building blocks using the NVIDIA CUDA technology platform. Technical report, University of Basel, 2007.
11. S. Conte and C. De Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill Higher Education, 1980.
12. F. Deserno, G. Hager, F. Brechtefeld, and G. Wellein. Basic optimization strategies for cfd-codes. Technical report, Regionales Rechenzentrum Erlangen, 2002.
13. P. Ezzatti and M. Fossati. Mejora del desempeño computacional del modelo RMA-11. Technical report, Universidad de la República, Uruguay, 2009.
14. P. Ezzatti and I. Piedra-Cueva. Mejora del desempeño computacional del RMA-10. In *Proceedings of VIII Congreso Argentino de Mecánica Computacional, Argentina*, 2005.
15. Z. Feng and P. Li. Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 647–654, Piscataway, NJ, USA, 2008. IEEE Press.
16. R. Fernando, editor. *GPU gems*. Addision-Wesley, Boston, 2004.
17. J. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, Berlin, 1999.
18. M. Fossati and I. Piedra-Cueva. Modelación tridimensional de la circulación en el Río de la Plata. In *Proceedings of XXII Congreso Latinoamericano de Hidráulica, Ciudad Guayana, Venezuela*, 2006.
19. D. Göddeke and R. Strzodka. Performance and accuracy of hardware-oriented native, emulated- and mixed-precision solvers in FEM simulations (part 2: Double precision GPUs). Technical report, Fakultät für Mathematik, Technische Universität Dortmund, 2008.
20. D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computer Systems Science and Engineering*, 4(1):36–55, 2008.

21. D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):221–256, 2007.
22. N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 102–111, Aire-la-Ville, Switzerland, 2003. Eurographics Association.
23. J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida, USA*, pages 1–7, 2008.
24. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.
25. nVidia. CUDA website. http://`www.nvidia.com/object/cuda_home.html`, 2010. Accessed on January 2010.
26. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
27. M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings of the IASTED International Conference on Visualization, Imaging and Image Processing, Marbella, Spain*, pages 193–202, 2001.
28. O. Schenk and K. Gärtner. Sparse factorization with two level scheduling in PARDISO. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, Virginia, 2001.
29. T. S. Sørensen and J. Mosegaard. An introduction to GPU accelerated surgical simulation. In M. Harders and G. Szekely, editors, *Third International Symposium on Biomedical Simulation*, volume 4072 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2006.
30. H. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM Journal of Numerical Analysis*, 1(5):530–558, 1968.
31. G Usera, A. Vernet, and J. Ferré. A parallel block-structured finite volume method for flows in complex geometry with sliding interfaces. *Flow, Turbulence and Combustion*, 80(3):346–350, 2008.
32. Y. Zhang, J. Cohen, and J. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 127–136, New York, NY, USA, 2010. ACM.