On the performance of a 2D unstructured computational rheology code on a GPU

Simão P. Pereira, Kees Vuik, Fernando T. Pinho, and João M. Nóbrega

Citation: AIP Conference Proceedings **1526**, 72 (2013); doi: 10.1063/1.4802604
View online: http://dx.doi.org/10.1063/1.4802604
View Table of Contents: http://scitation.aip.org/content/aip/proceeding/aipcp/1526?ver=pdfcov
Published by the AIP Publishing

On the Performance of a 2D Unstructured Computational Rheology Code on a GPU

Simão P. Pereira[a], Kees Vuik[b], Fernando T. Pinho[c], João M. Nóbrega[a]

[a]*I3N-Institute for Polymers and Composites, University of Minho, Campus Azurém, Guimarães 4800-058, Portugal*
[b]*Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Department of Applied Mathematics, Mekelweg 4, 2628 CD, Delft, The Netherlands*
[c]*CEFT, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal*

Abstract. The present work explores the massively parallel capabilities of the most advanced architecture of graphics processing units (GPUs) code named "Fermi", on a two-dimensional unstructured cell-centred finite volume code. We use the SIMPLE algorithm to solve the continuity and momentum equations that was fully ported to the GPU. The benefits of this implementation are compared with a serial implementation that traditionally runs on the central processing unit (CPU). The developed codes were assessed with the bench-mark problems of Poiseuille flow, for Newtonian and generalized Newtonian fluids, as well as by the lid-driven cavity and the sudden expansion flows for Newtonian fluids. The parallel (GPU) code accelerated the resolution of those three problems by factors of 19, 10 and 11, respectively, in comparison with the corresponding CPU single core counterpart. The results are a clear indication that GPUs are and will be useful in the field of computational fluid dynamics (CFD) for rheologically simple and complex fluids.

Keywords: Graphics processing units, Computational fluid dynamics, Parallelization, Finite volume method, Unstructured meshes.
PACS: 46.15.-x, 47.11.Df

INTRODUCTION

The demands to solve more complex and large dimensional problems has led to the development of parallel computing methodologies, aiming to solve the problems of interest within acceptable computational times. Traditionally, scientific computations are performed on Central Processing Units (CPU) either on sequential [1] or parallel [2,3] approaches, however the enhanced performance offered by the Graphics Processing Units (GPUs) is decisively contributing to its adoption in highly intensive scientific computations, with concomitant reductions of computation times [4,5]. Considering the fact that a couple of years ago GPUs were mainly targeted to the games industry, nowadays, performing heavy computations on GPUs has been defined as a new trend in computational science fields, due to the massively parallelism available on such devices [5].

Over the past five years, the number of non-graphical applications that use GPUs to perform heavy computations has significantly increased in a variety of fields [6,7]. However, as reported recently [4] there are some performance limitations in regard to

Novel Trends in Rheology V
AIP Conf. Proc. 1526, 72-89 (2013); doi: 10.1063/1.4802604

Computational Fluid Dynamics (CFD). Nevertheless, several implementations of CFD codes on GPUs have been reported and are briefly outlined hereafter.

Regarding the solution of the Euler or Navier-Stokes equations on GPU, one of the first implementation for compressible fluids can be found in Hagen et al. [8], indicating speed-ups of up to 25x, when compared with the corresponding single core CPU code. Brandvik and Pullan [9] reported speed-ups of 29x (2D) and 16x (3D) when solving the Euler equations using structured grids. Elsen et al. [7] also solved the Euler equations for hypersonic flows with speed-ups of 40x (simple geometries) or 20x (complex geometries). However, all these performances were attained by using single precision (SP) codes, because the first generation of GPU hardware did not allow double precision (DP) computations. Mixed precision techniques were used to compensate this problem as reported by Goddeke et al. [10], without major loss of precision.

The first DP incompressible Navier-Stokes solver for structured grids, achieved speed-ups of 8.5x when compared to the CPU implementation [11]. Corrigan et al. [12] reported speed-ups of 7.4x using DP with an unstructured cell-centred mesh finite volume code (FVM), for solving the Euler equations. The most relevant speed-ups were obtained on the implementation of an unstructured mesh vertex-based FVM code developed by Kampolis et al. [13], where maximum speed-ups of 28x (single precision), 25x (mixed precision) and 20x (double precision) for 2D and 3D Navier-Stokes solvers, were achieved. Subsequent work by the same group [14] improved their code and a maximum speed-up of 46x was achieved with the mixed precision technique, when solving unsteady laminar and turbulent flows.

In this work, we present a parallel Navier-Stokes type solver on GPU able to model generalized Newtonian fluids targeted to computational rheology applications. The aim is to evaluate its performance with inelastic models to move towards more complex constitutive equations. This will enable to tackle current computationally demanding viscoelastic fluids problems. Regarding the code implementation, no sort of optimization was introduced in the implementation with the purpose of evaluating if graphics cards required such robust programming skills. The parallel code was compared with a CPU version running on a single core.

GOVERNING EQUATIONS AND NUMERICAL METHOD

The governing equations for isothermal flow of an incompressible fluid are the continuity and momentum equations of motion written in vector form as

$$\nabla \cdot u = 0 \tag{1}$$

$$\rho \frac{\partial u}{\partial t} + \rho u \cdot \nabla u = -\nabla p + \nabla \cdot \tau \tag{2}$$

where ρ is the fluid density, u the velocity vector, p the pressure and τ the extra stress tensor. The simplest constitutive equation for incompressible fluids is the purely viscous generalized Newtonian fluid model of Eq. 3. If this equation is substituted into

in the momentum equation the Navier-Stokes equation for variable viscosity fluids is obtained [15].

$$\tau = \eta(\dot{\gamma})\dot{\gamma} \tag{3}$$

In Eq. 3 η is the viscosity and $\dot{\gamma}$ is the rate of strain tensor defined in Eq. 4. This model is valid for Newtonian and generalized Newtonian fluids, i.e., fluids with variable viscosity without any elastic effects. For Newtonian fluids the function $\eta(\dot{\gamma})$ takes on a constant value, while for the generalized Newtonian fluids the shear viscosity depends on an invariant rate of strain tensor [16], such as the shear rate of equation Eq. 5.

$$\dot{\gamma} = \nabla u + \nabla u^T \tag{4}$$

$$|\dot{\gamma}| = \sqrt{\frac{\dot{\gamma} : \dot{\gamma}}{2}} \tag{5}$$

The Bird-Carreau viscosity equation is a commonly employed model to account for the shear-thinning behaviour of polymer melts and solutions, and is given by

$$\eta(\dot{\gamma}) = \eta_\infty + \frac{\eta_0 - \eta_\infty}{\left(1 + (\lambda|\dot{\gamma}|)^2\right)^{\frac{1-n}{2}}} \nabla u + \nabla u^{\mathrm{T}} \tag{6}$$

which uses four empirical parameters. The model accounts for the low and high shear rate Newtonian plateau (η_0 and η_∞, respectively), along with the variable viscosity quantified by the power law exponent (n) and its onset at a shear rate of λ^{-1} [16].

The numerical solution of the governing equations are obtained by employing the finite volume method, used to solve flow problems with Newtonian and non-Newtonian fluids. In this method the computational domain is mapped onto control volumes over which the governing equations are initially integrated in space and time and benefiting from the application of the Gauss' Divergence Theorem. Then, the ensuing exact solutions are discretized to form a system of algebraic equations that can be solved using an iterative method [15,17], which includes a procedure to ensure the coupling between velocity and pressure fields. At the end, any of the discretized governing equations can be written in the following general linear form at any cell P for the property ϕ (u for instance)

$$a_{\mathrm{P}}\phi_{\mathrm{P}} - \sum_{nb} a_{nb}\phi_{nb} = S_{\phi} \tag{7}$$

in which contributions to the central and neighbour cells are carried out through the corresponding coefficients. The specific expressions depend on the adopted differencing scheme. In this work, the discretization of the governing equations was made using an unstructured two-dimensional mesh based on triangular control volumes [17]. We adopt the cell-centred scheme, in which all variables are computed

at the centre of the control volume, also known as collocated arrangement. The advantage of using unstructured meshes, in comparison with structured meshes, lays on the fact that more complex geometries can be mapped without the need for unnecessary mesh refinements elsewhere [17,18].

Considering the discretization of the momentum equation we use a central-difference scheme to determine the diffusive term, a first-order upwind or a second-order MUSCL scheme [19] to compute the convective term and the pressure gradient is computed explicitly based on the least-square gradient technique [17]. The time discretization is by the first-order Euler implicit scheme. When performing steady state calculations we use this term to march the computation along time, but without converging at each time step (pseudo-transient approach), as an alternative to under-relaxation [17].

To overcome the nonlinearities of the governing equations, their coefficients are always computed on the field data obtained in the previous iteration. To satisfy both the mass and momentum equations, and since there is no specific equations for the pressure, the SIMPLE algorithm [20] is adopted, in which the continuity equation is transformed into an equation for a pressure correction. The idea behind this algorithm is to start with a guessed velocity field in order to compute the convective fluxes at the cell faces, and a guessed pressure field to solve the momentum equations, obtaining a new velocity field. The new velocity field does not satisfy continuity, but the pressure correction equation, derived from the continuity equation, is then solved to correct both the pressure and velocity fields, thus obtaining a mass conservative velocity field. Since pressure and velocities are computed and stored at the cell centres, a special interpolation scheme has to be used when calculating the mass fluxes at cell faces in order to ensure complete coupling between the two fields. The Rhie and Chow method [21] is used for this purpose. The algorithm employed is illustrated in Figure 1 along with details regarding the flow chart of the parallel (GPU) implementation that will be mentioned later.

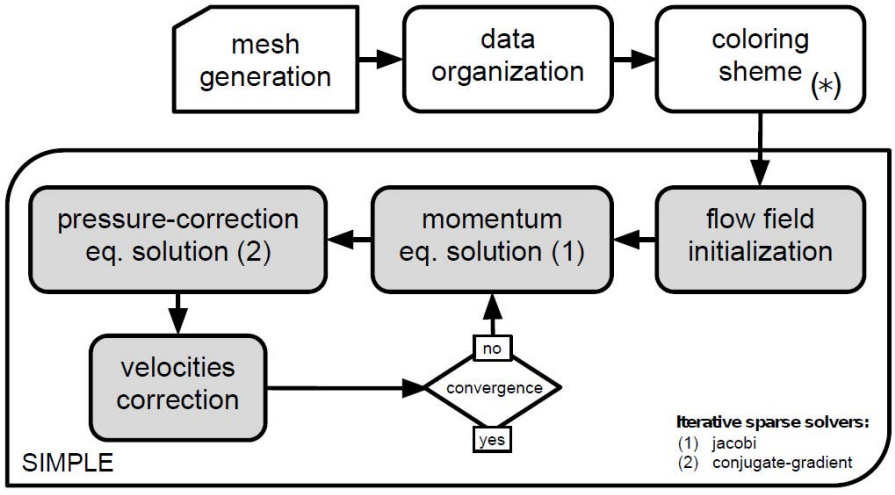

FIGURE 1. Flow chart of the implemented SIMPLE algorithm. The coloring scheme routine signed with (∗), is used just for the parallel implementation with evidence of the SIMPLE algorithm. The grey boxes correspond to the routines that were ported to the GPU, in the parallel of the code.

The process converges when sufficient velocity and pressure corrections are performed, in order to satisfy the chosen stopping criteria for the SIMPLE algorithm. The convergence criterion of SIMPLE is based on the normalized residuals of the momentum and pressure equations, where the maximum residual is taken to be the maximum of the first 5 iterations. The normalized residuals must fall below 10^{-4} for momentum and pressure-correction equations.

IMPLEMENTATION ON THE GPU

In this section the GPU computing environment is briefly introduced and implementation details of the developed numerical modelling code are presented.

Graphics Processing Units

GPUs are now considered to be highly parallel multi-core processors with large floating point performance peak (GFLOPS) and higher memory band- width (GB/s) than CPUs. Recent developments on the architecture of graphics cards introduced the Fermi architecture, considered to be the first GPU suitable for scientific computations, since the previous generations of GPU architectures lacked some important features (e.g. double-precision support, extended memory hierarchy, handling of algorithms with irregular patterns, among others) in order to be suitable for a wide range of scientific computations [22].

Basically, the Fermi architecture features up to 512 CUDA cores or Streaming Processors (SPs) organized in Streaming Multiprocessors (SMs) each containing 32 cores. The GPU has its own DRAM memory that can be as large as 6 GB depending on the graphic card, and exchanging data between the host (CPU) and the device (GPU) memories is done via a PCI-Express interface. On the device (GPU) there are different types of memories, from fast to slow, that can be used and controlled by the programmer. The main advantage of the Fermi architecture is the existence of an extended memory hierarchy that caches memory accesses to global memory automatically, i.e. without the programmers' intervention, a feature that enables algorithms with random memory accesses to be suitable for GPUs, as opposed to previous architectures [23].

Execution Model

The Compute Unified Device Architecture (CUDA) introduced by NVIDIA [23] is a general purpose parallel computing architecture that provides a C/C++ language interface to the hardware, based on a scalable programming model and on an instruction set architecture. In other words, CUDA is simultaneously a hardware and a software architecture that enables the execution of parallel programs on GPUs [22,24]. The device (GPU) is considered to be a host (CPU) co-processor and device function calls for device memory allocation, data transfer between host and device and kernel calls are controlled by the host. Kernels are device functions that execute the same instruction in parallel for a set of threads organized in blocks, which compose a grid of one or more blocks. Essentially each block is an amount of work that is assigned to

each SM and the threads within the block are executed simultaneously by all SPs in each SM. Increasing the number of cores and of SM allows to solve more blocks of work concurrently, enabling an automatic scalability with the number of CUDA cores [23]. In the Fermi architecture, each SM schedules threads in groups of 32 parallel threads called warps, which are the minimum size of the data processed. The interested reader should consult [23,24] for more details on GPU architecture and programming.

Implementation Issues

The parallel implementation of the code on a GPU is not a hard task, but getting performance benefits from the parallel implementation is not so straightforward. The most basic strategy is to remove the bottlenecks of the serial code, by porting to the GPUs the most time consuming routines. This must be done carefully to avoid unnecessary memory transfers between host and device. Sometimes it compensates to run part of the code on the GPU rather than on the CPU, even if that does not show performance gains, just to avoid data transfer between host and device.

The flow chart of the code implemented on the GPU is shown in Figure 1 which indicates the routines that run on the GPU and on the CPU. As shown, some lightweight routines are executed only once, hence it is not worth to port them into the device. These are the mesh generation, the data structuring arising from the mesh and the colouring scheme that will be detailed later. These routines are required along the code and are all computed on the CPU, and the data copied to the GPU global memory.

An important issue regards the assembly of the algebraic system of equations on parallel architectures. The system of equations is assembled by sweeping through all edges of the computational mesh in order to collect contributions from each edge to both triangular control volume straddling that edge. However in parallel architectures it is necessary to adopt a special scheme to avoid the occurrence of race conditions [13,26], which occur when two or more threads (running in parallel) try to access simultaneously the same memory address, leading to information loss. One way to avoid this problem in parallel architectures is to adopt a colouring scheme when assembling the system of equations [13,26]. In this work we coloured those edges that have independent diagonal contribution to the matrix which guarantees that each triangle receives only one contribution from each edge colour, i.e. this assumes that implicit contributions to the diagonal are not carried out in the same assembly step. In the case of triangular control volumes, six colouring steps (3 per cell) are necessary to assemble the system of equations, since every edge has two adjacent cells. The routine to colour the edges is per- formed once on the host and the information is further copied to the device memory.

The iterative methods for solving sparse linear systems used in this work were the point-iterative Jacobi and the conjugate-gradient, which use a matrix stored in the compressed sparse row (CSR) format [27]. Whenever possible, the GPU parallel code uses the CUBLAS and CUSPARSE libraries from NVIDIA's toolkit [23], in routines such as sparse matrix-vector multiplication ($y = Ax + y$), vector-scalar multiplications and vector update or dot product. The convergence criteria or accuracy of the iterative

solvers was defined by the ratio of the current and initial normalized residuals $\left(\|b-Ax\|_2/\|b-Ax_0\|_2\right)$. The key feature of sparse matrix-vector multiplication operations is the large number of load instructions relative to the floating point instructions and this is the reason why they are considered to be a memory bound operation.

CODE ASSESSMENT

In this section the numerical solution of the benchmark problems used to assess the implementation of the GPU parallel code are presented. All computations were performed with double-precision and meshes were generated with the open source software Gmsh [28].

Poiseuille Flow between Parallel Plates

In Poiseuille flow, the flow is driven by a streamwise pressure gradient in equilibrium with the viscous drag. For Newtonian and some non-Newtonian fluids there is a full analytical solution for the fully developed velocity and pressure fields, which depends on the shear viscosity of the fluid. In the case of a 2D parallel plate flow, the length L and width of the channel are assumed to be much larger than the thickness H of the channel as depicted in Figure 2(a). Figure 2(b) shows one of the meshes used (PF01) having a cell thickness of the order of $0.033H$, since a variety of computational meshes (detailed in Table 1) were used to evaluate how the performance scales with the number of cells, as will be shown in the next section. All meshes are unstructured with a "quasi-uniform" distribution of cell sizes, i.e., no attempt was made to refine them (see Figure 2). The ratio between length (L) and thickness (H) of the channel was defined to be 5.

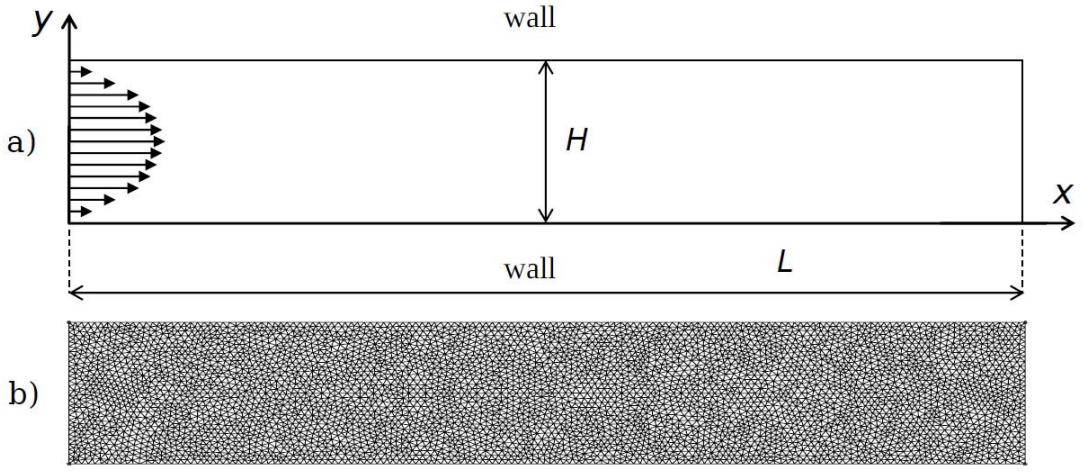

FIGURE 2. Poiseuille flow problem (a), boundary conditions and representative mesh (b).

TABLE 1. Computational meshes used for the Poiseuille flow case study.

Meshes	PF01	PF02	PF03	PF04	PF05	PF06	PF07	PF08
Cells	$1.0{\cdot}10^4$	$2.8{\cdot}10^4$	$5.6{\cdot}10^4$	$7.2{\cdot}10^4$	$1.4{\cdot}10^5$	$2.0{\cdot}10^5$	$3.3{\cdot}10^5$	$4.6{\cdot}10^5$
$(\Delta x/H)_{\min}$	0.033	0.02	0.014	0.013	0.0091	0.0077	0.0059	0.005

The problem was solved considering a Newtonian fluid flow at Re = 0.1, where the Re is computed based on the mean fluid velocity, channel thickness and the kinematic viscosity (η). A fully developed velocity profile was used as inlet boundary condition and as outlet a zero normal gradient for all flow variables was assumed. In this problem, the first-order UDS scheme was used to estimate the convective fluxes and both the momentum and pressure-correction were solved with an accuracy of 10^{-2}. A time-step of 10^{-4} s was used for the time-marching computations and pressure corrections were relaxed with a constant value of 0.1.

The non-Newtonian model used in this test is that of a purely viscous fluid, the generalized Newtonian fluid model, with the viscosity law given by the Bird-Carreau equation (Eq. 6). In this case the zero shear viscosity is used to compute the Reynolds number. The computations were carried out with different values for the power index n (0.4 − 1.0) and the parameters used for the Bird-Carreau model were: $\eta_0 = 1.3{\cdot}10^4$ Pa.s, $\eta_\infty = 1.0{\cdot}10^{-3}$ Pa.s and $\lambda = 5.4{\cdot}10^{-2}$ s. The dimensionless velocity profiles computed with the GPU code and using the computational mesh PF01 (see Table 1) for both Newtonian and generalized Newtonian fluids are presented in Figure 3. Since there is no analytical solution for the Bird-Carreau constitutive equation, the numerical predictions by the developed GPU code were compared with results from a reliable and well-verified finite volume code [1], that uses structured meshes.

The results in Figure 3 show a good agreement between both codes. As expected, by decreasing the power index (n) the velocity profile tends to a plug shape, the typical behaviour for a shear-thinning fluid.

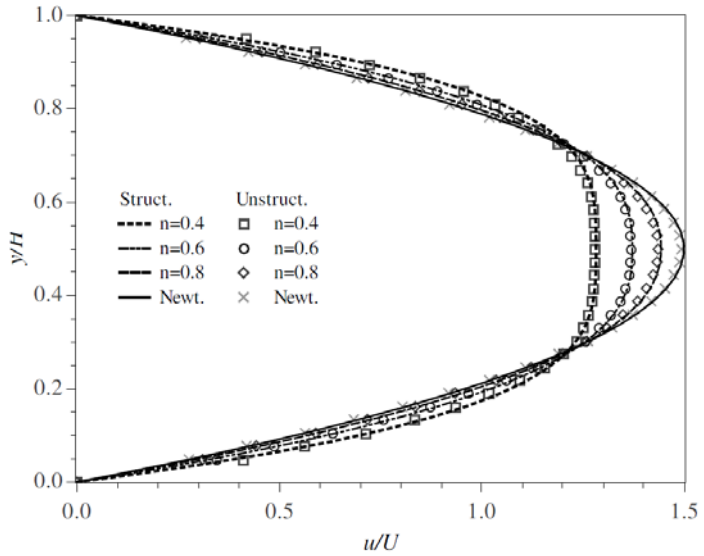

FIGURE 3. Dimensionless velocity profiles for Newtonian and Bird-Carreau fluids at Re = 0.1, obtained both with a previously assessed code [1] (struct.) and the developed code (unstruct.).

Lid-Driven Cavity Flow

The lid-driven cavity flow is a benchmark problem usually employed to verify the Navier-Stokes equations solvers and for the assessment of numerical techniques and accuracy. The problem considers incompressible flow in a square cavity with side length H, where an upper lid moves with a known velocity (U), as illustrated in Figure 4.

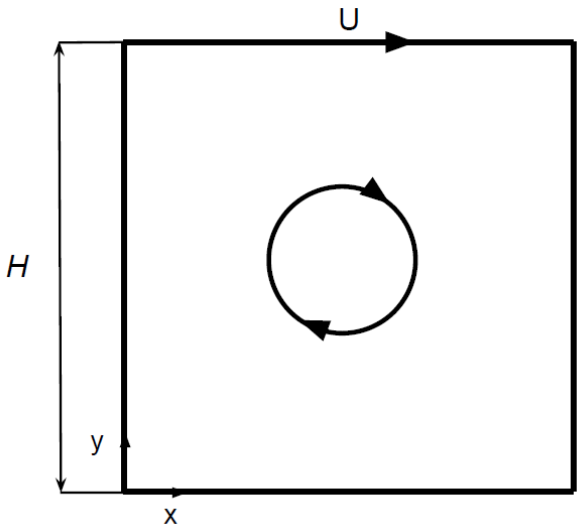

FIGURE 4. Description of lid-driven cavity flow problem.

This test problem is well documented in the literature [29,30,31], using different solution procedures and Reynolds numbers ranging from 100 to 10,000. The major difficulty with this problem is to capture the flow near the corners, where vortices appear. Here, numerical solutions for the cavity flow at Re = 1,000, are presented and compared with steady state benchmark quality data from the literature. The solutions were obtained by using the MUSCL scheme to estimate the convective fluxes. All the computations were performed using the pseudo-transient approach, considering that the fluid was initially at rest. The square cavity height was set to $H = 0.1$ m and the top wall moves at a speed of $U = 0.01$ m.s^{-1}, corresponding to flow at Re =1,000. The time-step of 0.005 s and a constant value of 0.5 were used for the time-marching computations and relaxation of pressure correction, respectively. Furthermore, the accuracy of the iterative solvers was defined to be of 10^{-1}.

The assessment work was performed using mesh LD01 (see Table 2). The numerical results obtained were assessed by means of velocity plots and streamlines. The computed u-velocity along the central vertical centre line and the v-velocity along the horizontal centre line are compared with the values reported in [29]. The velocity profiles are plotted in Figure 5 and match very well with the reference values reported in the literature. For assessment purposes it is also useful to present the flow field using streamlines, because it enables the visualization of the global flow field. They are plotted in Figure 6, and were obtained using the Paraview's [32] implementation of the fourth-order Runge-Kutta scheme on the Stream Tracer filter.

TABLE 2. Computational meshes used in the lid-driven cavity flow case study.

Meshes	**LD01**	**LD02**	**LD03**	**LD04**	**LD05**	**LD06**	**LD07**	**LD08**
Cells	$5.40{\cdot}10^3$	$2.39{\cdot}10^4$	$5.17{\cdot}10^4$	$9.53{\cdot}10^4$	$1.43{\cdot}10^5$	$2.10{\cdot}10^5$	$3.69{\cdot}10^5$	$6.91{\cdot}10^5$
$(\Delta x/H)_{\min}$	0.02	0.01	0.0067	0.005	0.004	0.0033	0.0025	0.0018

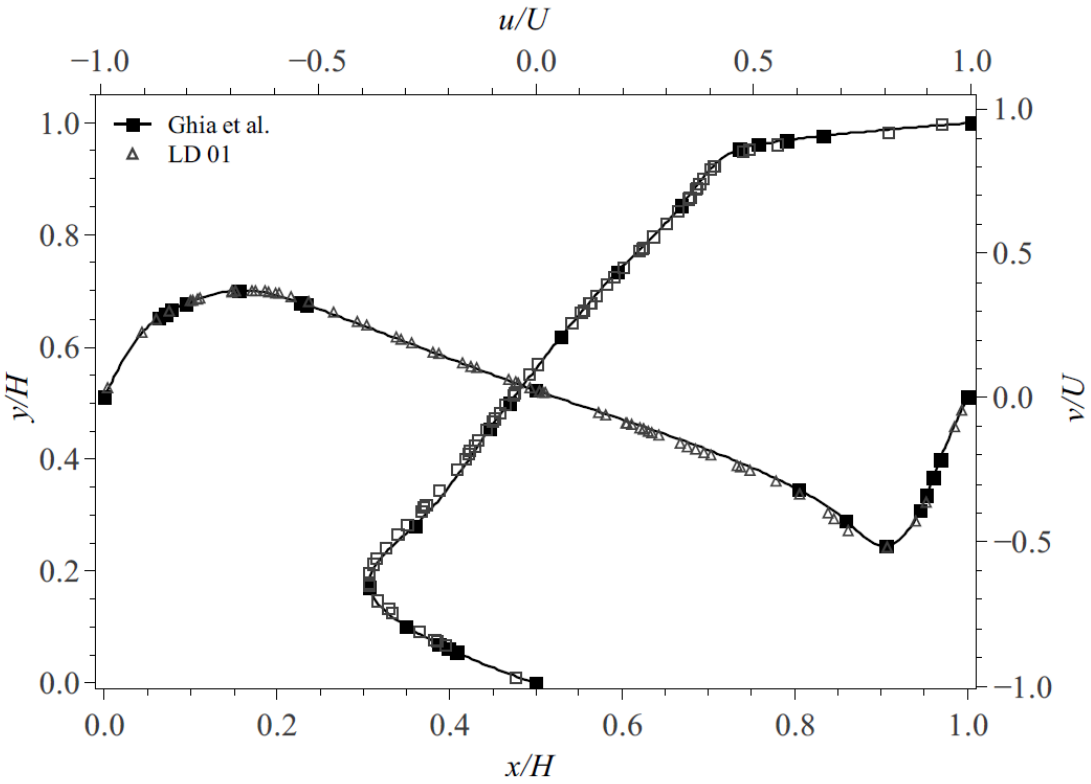

FIGURE 5. Velocity profiles for Re = 1,000 using mesh LD 01, for the lid-driven cavity problem.

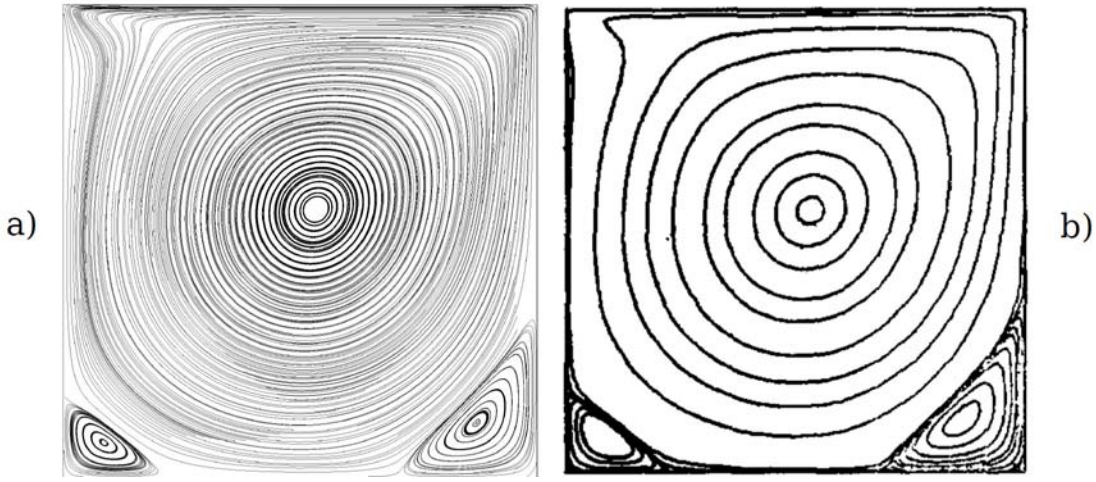

FIGURE 6. Streamlines for the lid-driven cavity problem at Re = 1,000: a) this code, b) adapted from Ghia et al. [29].

The streamlines predicted by the code match very well the literature data [29] including the shape and location of the vortices.

Sudden Expansion

The last problem employed for assessment purposes was the planar sudden expansion, whose geometry is illustrated in Figure 7. It consists of an upstream channel of thickness *H* followed by a downstream channel of thickness 4*H*. For this problem a Newtonian fluid was considered, assuming fully developed flow at the channel inlet. Due to the very long downstream channel (300*H*), the gradient of flow variables at the outlet is null, except for the pressure that is linearly extrapolated from the domain. This benchmark problem has been extensively used to verify numerical solutions and schemes [33,34,35] used in incompressible laminar flow of Newtonian and non-Newtonian fluids. At low Reynolds numbers the flow is symmetric and at a critical Reynolds number, that depends on expansion ratio, the flow becomes asymmetric [34], but remains steady. Using the developed code we solve numerically the plane sudden expansion following the geometry used by Drikakis [34] at Reynolds numbers of 55 and 80. As done in [34], the Reynolds number was computed based on the maximum velocity in the fully developed profile in the upstream channel. For this problem the meshes described in Table 3 were considered, and in particular mesh SE01 was employed here for assessment purposes. The numerical results obtained are presented as streamlines shown in Figure 8 for Re = 55, where the flow is still symmetric, and at Re = 80, where the transition from symmetric to asymmetric flow has already taken place. The similarity of these results with those of Drikakis by [34] for the same flow conditions, also shown in Figure 8, are an additional verification of the developed code.

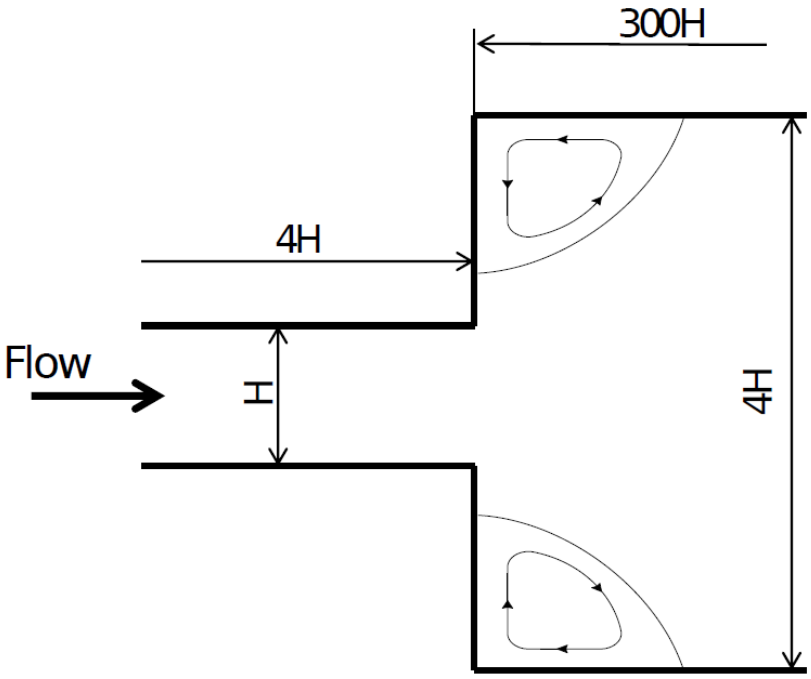

FIGURE 7. Geometry employed for the sudden expansion problem.

TABLE 3. Computational meshes used in the sudden expansion flow case study.

Meshes	SE01	SE02
Cells	$1.28 \cdot 10^5$	$3.85 \cdot 10^5$
$(\Delta x/H)_{min}$	0.045	0.033

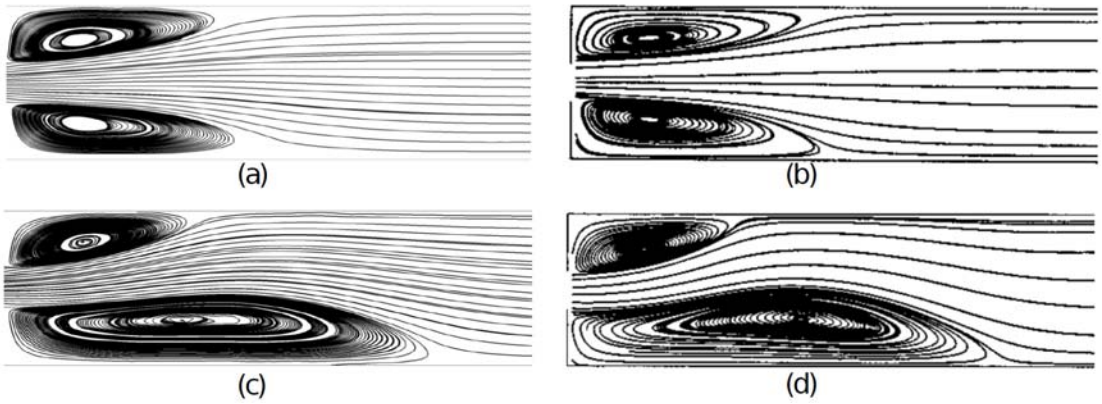

FIGURE 1. Streamline for the sudden expansion case study: results obtained with the developed code - (a) Re = 55 and (c) Re = 80 - and results adapted from Drikakis [34] - (b) Re = 55 and (d) Re = 80.

PERFORMANCE ANALYSIS

In this section a performance comparison between the serial and parallel implementations of the developed codes is carried out, based on mesh refinement scalability for the three benchmark flow problems presented before.

Methodology Employed for the Performance Evaluation

A performance comparison between the developed GPU numerical code and a quite similar CPU code running in a single core is presented in this section, with the goal of exploring the benefits of porting code to GPUs. The performance comparison is carried out in terms of the computational time consumed as a function of the number of computational cells (problem dimension).

Both numerical codes have a set of routines to solve flow problems which include the implementation of the SIMPLE algorithm, along with other pre- and post-processing routines essential to the developed numerical codes, such as mesh generation, handling of mesh data, computing geometrical properties, among others, which are not analyzed here because they are exactly the same in both codes. Instead, our focus is to those routines that are essential to the parallel implementation and which are listed in Table 4. The circles are used to indicate the routines that run fully on the GPU, whereas the crosses refer to those running on the CPU even on the GPU code. Note that the GPU implementation requires 3 additional routines that are executed only once, and do not exist in the CPU code. The routines for solving the momentum equation, the pressure-correction equation and the velocity corrections are invoked several times during the SIMPLE iterative process, and were programmed to run fully on the GPU. During this study it was also found that it is crucial to

implement the complete algorithm on the GPU, in order to minimize data transfers between host and device.

TABLE 4. Most important code routines for each code implementation.

Routine	CPU	GPU
Coloring scheme	—	X
Copy to Device	—	X
Solve momentum equations	X	O
Solve pressure-correction equation	X	O
SIMPLE corrections	X	O
Copy to Host	—	X

Time measurements of the most relevant routines presented later on were obtained with the command *gettimeofday()*, both for the CPU and GPU implementations. Since kernel calls are asynchronous to host threads, in the case of the GPU implementation the command *cudaThreadSynchronize()* must precede the timing command in order to guarantee that the GPU has finished the computations [23].

Poiseuille Flow between Parallel Plates

The details of the computational meshes were introduced in Table 1, thus we present in Table 5 the computational times taken by the SIMPLE algorithm for both CPU and GPU implementations, along with the duration of data transfers between host and device memories. The results show that the parallel implementation accelerates the SIMPLE algorithm up to 19x, which corresponds to a reduction of the computational time from 23 hr 22 min to 1 hr 12 min for the finest mesh.

As can be concluded from the data shown in Table 5 increasing the amount of work increases the speed-up of the parallel implementation, which emphasizes the advantages of GPU parallelization.

TABLE 5. Computational times and speed-up for the SIMPLE algorithm and data transfers, in the Poiseuille flow case study, for the CPU and GPU implementations.

Meshes	PF01	PF02	PF03	PF04	PF05	PF06	PF07	PF08
CPU (s)								
SIMPLE CPU	63.1	400.8	1343.7	2202.8	7807.5	16704.7	52015.0	84120.2
GPU (s)								
Copy to Device	0.0025	0.0054	0.0088	0.012	0.018	0.026	0.042	0.06
SIMPLE GPU	25.3	79.0	192.1	267.1	702.8	1243.9	3119.5	4334.7
Copy to Host	0.0003	0.0006	0.0011	0.0013	0.0025	0.0035	0.0057	0.0078
Speed Up								
Newtonian	2.5	5.1	7.0	8.2	11.1	13.4	16.7	19.4
GNF(n = 0.8)	2.4	5.0	6.9	7.9	11.1	13.7	-	-

It is well known that solving the system of equations is always the bottleneck of computational fluid mechanics codes. However, solving only the system of equations on the GPU is not the correct strategy, since data transfers between host and device are

very slow and easily kill the application performance. This justifies the complete implementation of the SIMPLE algorithm in the parallel architecture, in order to avoid data transfers between host and device at every iteration of the algorithm. Copying data to the device memory is performed once at the beginning of the SIMPLE algorithm and only again at the end of the algorithm, after convergence. This minimizes the impact of the data transfer, which can be neglected with respect to the computations, since its relative contribution varies from 0.01% to 0.001% for the problems studied.

Within each code (serial/CPU and parallel/GPU) the relative amount of time spent on each routine is similar as shown in Figure 9. Note that most of the time is actually devoted to solving the momentum equations and in particular in the *x*-direction, since this problem is mainly a one-dimensional flow. Usually, in this type of algorithms, solving the pressure-correction equation is more time consuming than solving the momentum equations, however this was not seen here because of the convergence rate of the Jacobi iterative solver used for the latter, that is lower than that of the conjugate-gradient method, used for the former.

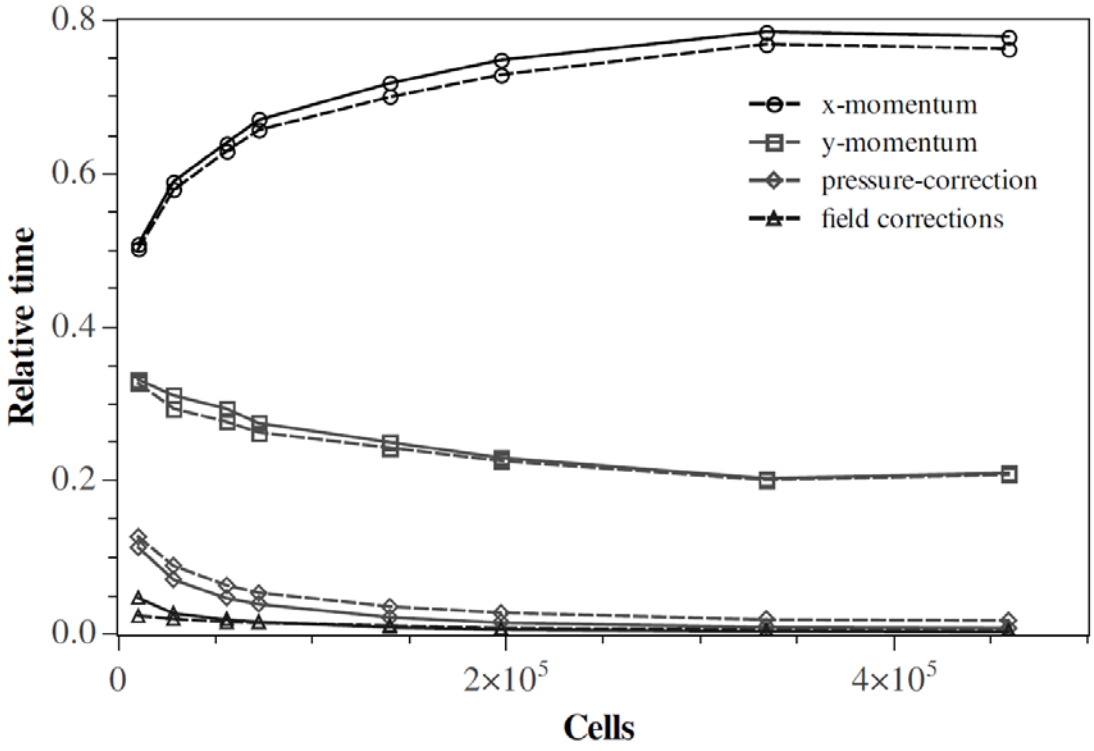

FIGURE 9. Relative timings for the Poiseuille flow problem. Full and dashed lines correspond to the CPU and GPU implementations, respectively.

A performance comparison was also carried out for the Bird-Carreau fluid computations, although not in such detail as for the Newtonian case. The results in terms of speed-up are plotted in Figure 10 and show that the generalized Newtonian model runs as fast as the Newtonian, showing the same performance trends.

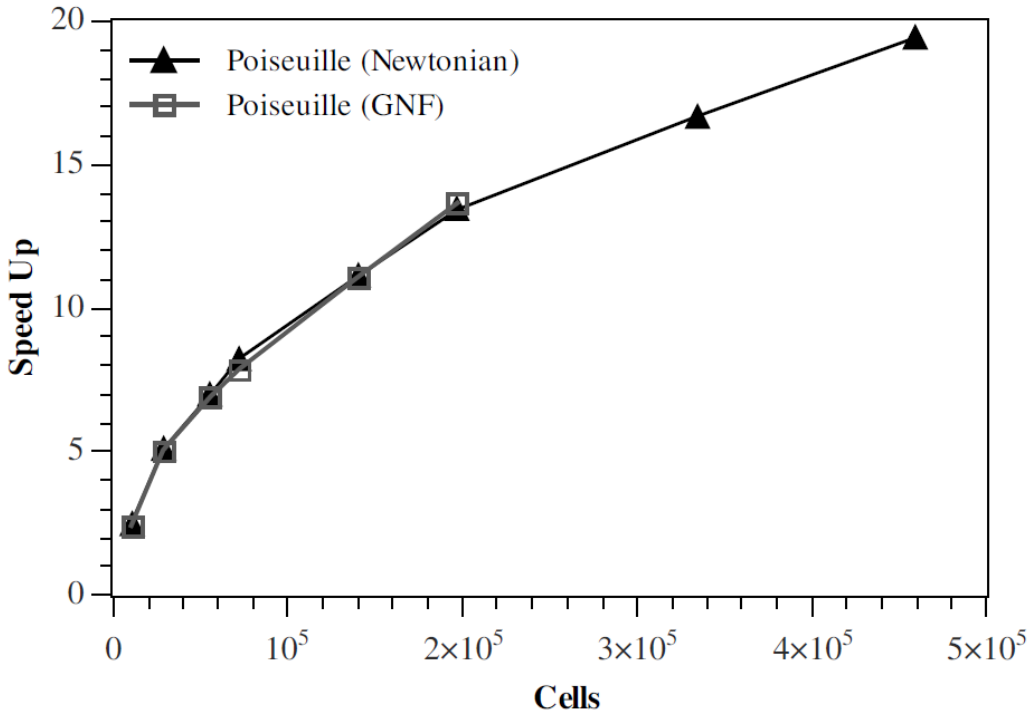

FIGURE 2. Speed-up for the Poiseuille flow for Newtonian and Bird-Carreau model ($n = 0.8$).

Indeed, these models only require the change of the local viscosity, according to the rate of deformation by using Eqs. 5 and 6. A maximum speed-up of approximately 14 was attained for the mesh PF06. So far, the performance results are quite impressive for such a simple implementation without of the usual referred, and time consuming for the code development, memory optimizations required to achieve impressive performance speed-ups in GPUs. [23].

Lid-Driven Cavity Flow

For this flow the computational times are those in Table 6 for the SIMPLE algorithm on the CPU and GPU with the meshes on Table 2. From now on, we ignore the data transfers between host and device since it was shown in the previous problem that these operations are relatively small in terms of computational time. This is mainly because the copying routines are executed only twice, once at the beginning and then at the end of the calculation.

TABLE 6. Computational time for the calculation with the SIMPLE algorithm and data transfers. Comparison between the CPU and GPU implementations for the lid-driven cavity flow case study.

Meshes	**LD01**	**LD02**	**LD03**	**LD04**	**LD05**	**LD06**	**LD07**	**LD08**
				timings (s)				
SIMPLE CPU	295.6	2050.1	6427.2	17541.9	36643.5	39702.5	77939.8	161107.6
SIMPLE GPU	177.5	509.0	1012.1	2451.6	4884.6	4972.1	8415.4	16444.5
				Speed Up				
	2.5	5.1	7.0	8.2	11.1	13.4	16.7	19.4

The maximum speed-up factor obtained for this case study was 10, which is approximately half those of the previous problems. In a typical computation the most time consuming routine is devoted to solving the system of equations, but here, we had to use a small time-step to under-relax properly the momentum equations in order to achieve convergence. As a result, the iterative solver required less iterations to achieve the desired accuracy. Smaller time-steps require more outer iterations on the SIMPLE algorithm and as a result the assembly of the system of equations limits performance to a maximum speed-up of 10.

Sudden Expansion

For the sudden expansion flow problem, we simply considered the two highly refined meshes in Table 3, one with 128k cells and the other with 385k cells. The speed-up factors obtained for the 1:4 planar sudden expansion at Re = 80 were 7.9 and 10.8, for the coarse and finest meshes, respectively, a clear indication that it follows the performance trend of the previous problem. It is important to note that there is a dependence of the performance on the simulation parameters or on flow conditions, since the relative time spent on each routine varies and some routines perform better than others when parallelized.

CONCLUSION

This work presents and discusses the GPU parallel implementation and performance scalability of a 2D finite-volume flow solver for unstructured meshes. The performance assessment is carried out in comparison with a serial version of the code running on a single CPU core. For the GPU version of the developed numerical code, the latest Fermi graphics cards architecture was used, which dispenses with the need for the programmer to devote most of the developing time in memory optimizations to obtain relevant performance gains. The performance benefits obtained with the GPU implementation relative to the single core CPU code are quantified in maximum speed-up factors of 19, 10 and 11, for the Poiseuille channel, lid-driven cavity and 1:4 planar sudden expansion flows, respectively.

Considering the GPU availability, cost and easiness to program, the results obtained are an indication that the full exploitation of the capabilities of these devices can be very useful for future computational fluid dynamics and computational rheology calculations. Porting numerical code to the massively parallel graphics cards compensates when the amount of work is large. The parallel code runs considerably faster that its serial counterpart, even without any memory optimizations that can speed-up even further the computations.

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding by FEDER via Fundação para a Ciência e Tecnologia through projects FCOMP-01-0124-FEDER-015126 (Ref. FCT PTDC/EMEMFE/ 113988/2009) and PEst-C/CTM/LA0025/2011 (Strategic Project - LA 25 2011-2012).

REFERENCES

1. P. Oliveira, F. Pinho and G. Pinto, *J. Non-Newtonian Fluid Mech.* **79**, 1-43 (1998).
2. P. F. Fischer and A. T. Patera, *Annu. Rev. Fluid Mech.* **26**, 483-527 (1994).
3. H.-S. Dou and N. Phan-Thien, *Comput. Mech.* **30**, 265-280 (2003).
4. J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn and T. Purcell, *Comput. Graph. Forum* **26**, 80-113 (2007).
5. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, *Proceedings of the IEEE* **96**, 879-899 (2008).
6. J. A. Anderson, C. D. Lorenz and A. Travesset, *J. Comput. Phys.* **227**, 5342-5359 (2008).
7. E. Elsen, P. LeGresley and E. Darve, *J. Comput. Phys.* **227**, 10148-10161 (2008).
8. T. R. Hagen, K.-A. Lie and J. R. Natvig, *Lecture Notes in Computer Science* **3994**, 220-227 (2006).
9. T. Brandvik and G. Pullan, "Acceleration of a 3D Euler solver using commodity graphics hardware", in *46th AIAA Aerospace Sciences Meeting* (January 2008).
10. D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker and S. Turek, *Int. J. Comput. Sci. Eng.* **4**, 36-55 (2008).
11. J. Cohen and J. Molemake, "A fast double precision CFD code using CUDA", in *21st International Conference on Parallel Computational Fluid Dynamics* (May 2009).
12. A. Corrigan, F. F. Camelli, R. Lohner and J. Wallin, Running unstructured grid–based CFD solvers on modern graphics hardware, *International Journal for Numerical Methods in Fluids* **66**, 221-229 (2011).
13. I. Kampolis, X. Trompoukis, V. Asouti and K. Giannakoglou, *Comput. Method. Appl. Mech. Eng.* **199**, 712-722 (2010).
14. V. G. Asouti, X. S. Trompoukis, I. C. Kampolis and K. C. Giannakoglou, *Int. J. Numer. Meth. Fl.* **67**, 232-246 (2010).
15. J. Blazek, *Computational Fluid Dynamics: Principles and Applications (2nd edition)*, Amsterdam: Elsevier Science, 2006.
16. Z. Tadmor and C. Gogos, *Principles of Polymer Processing (2nd edition)*, New Jersey: Wiley-Interscience, 2006.
17. H. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics the Finite Volume Method (2nd edition)*, Harlow: Pearson Education Limited, 2007.
18. L. Sun, S. Mathur and J. Murthy, *Numer. Heat Trans., Part B: Fundamentals* **58**, 217-241 (2010).
19. M. Darwish and F. Moukalled, *Int. J. Heat Mass Tran.* **46**, 599-611 (2003).
20. S. Patankar, *Numerical Heat Transfer and Fluid Flow*, New York: Taylor & Francis, 1984.
21. C. Rhie and W. Chow, *AIAA J.* **21**, 1525-1532 (1982).
22. NVIDIA, Fermi compute architecture whitepaper, http://www.nvidia.com/ob ject/fermi-architecture, 2009.
23. NVIDIA, CUDA C Programming Guide Version 4.0, http://www.developer.nvidia.com/ nvidia-gpu-computing- documentation, 2011.
24. D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (1st edition)*, Burlington: Morgan Kaufmann, 2010.
25. NVIDIA, CUDA C Programming Best Practices Guide Version 4.0, http://www.developer.nvidia.com/nvidia-gpu-computing- documentation, 2011.
26. C. Cecka, A. J. Lew and E. Darve, *Int. J. Numer. Meth. Eng.* **85**, 1-6 (2010).

27. Y. Saad, *Iterative Methods for Sparse Linear Systems (2nd edition)*, Society for Industrial and Applied Mathematics, 2003.
28. C. Geuzaine and J. -F. Remacle, *Int. J. Numer. Meth. Eng.* **79**, 1309-1331 (2009).
29. U. Ghia, K. Ghia and C. Shin, *J. Comput. Phys.* **48**, 387-411 (1982).
30. O. Botella and R. Peyret, *Comput. Fluids* **27**, 421-433 (1998).
31. C. Bruneau and M. Saad, *Comput. Fluids* **35**, 326-348 (2006).
32. A. Henderson, ParaView User's Guide (v3.10), Kitware, Inc, 2011.
33. F. Durst, A. Melling and J. H. Whitelaw, *J. Fluid Mech.* **64**, 111-128 (1974).
34. D. Drikakis, *Phys. Fluids* **9**, 76-87 (1997).
35. N. Panagiotis, *J. Non-Newtonian Fluid Mech.* **133**, 132-140 (2006).