View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

UPCommons

Portal del coneixement obert de la UPC

http://upcommons.upc.edu/e-prints

Aquest és un manuscrit acceptat d'un article publicat per Taylor & Francis a *International journal of computational fluid dynamics* el **28/09/2017**, disponible en línia: https://www.tandfonline.com/doi/full/10.1080/10618562.2017.1390084

This is an Accepted Manuscript of an article published by Taylor & Francis in *International journal of computational fluid dynamics* on **28/09/2017**, available online: https://www.tandfonline.com/doi/full/10.1080/10618562.2017.1390084

To appear in the *International Journal of Computational Fluid Dynamics*
Vol. 00, No. 00, 00 Month 20XX, 1–20

Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers

G. Oyarzun[a], R. Borrell[a,b], A.Gorobets[a,c] and A. Oliva[a]*

[a] *Heat and Mass Transfer Technological Center, ETSEIAT, Technical University of Catalonia, Terrassa, Spain*; [b] *Termo Fluids, S.L., Sabadell (Barcelona), Spain*; [c] *Keldysh Institute of Applied Mathematics, Moscow, Russia*

(*Received 00 Month 20XX; final version received 00 Month 20XX*)

Nowadays HPC systems experience a disruptive moment with a variety of novel architectures and frameworks, without any clarity of which one is going to prevail. In this context, the portability of codes across different architectures is of major importance. This paper presents a portable implementation model based on an algebraic operational approach for DNS and LES of incompressible turbulent flows using unstructured hybrid meshes. The strategy proposed consists in representing the whole time-integration algorithm using only three basic algebraic operations: sparse matrix-vector product, a linear combination of vectors and dot product. The main idea is based on decomposing the non-linear operators into a concatenation of two SpMV operations. This provides high modularity and portability. An exhaustive analysis of the proposed implementation for hybrid CPU/GPU supercomputers has been conducted with tests using up to 128 GPUs. The main objective consists in understanding the challenges of implementing CFD codes on new architectures.

Keywords: Portability, Heterogeneous Computing, GPU, MPI+CUDA, OpenCL, sliced ELLPACK, CFD code

1. Introduction

The pursuit of exascale has driven the adoption of new parallel models and architectures into HPC systems in order to bypass the power limitations of the multi-core CPUs. The last prominent trend has been the introduction of massively-parallel accelerators used as math co-processors to increase the throughput and the FLOPS per watt ratio of the HPC systems. Such devices exploit a stream processing paradigm that is closely related to single instruction multiple data (SIMD) parallelism of the vector registers in the modern multi-core CPUs. The problem that the CFD community has to face is the variety of competing architectures and frameworks without any clarity of which one is going to prevail. This level of uncertainty and the complexity of porting scientific codes make the decision of adopting a specific computing paradigm a risky decision. The traditional anatomy of an incompressible CFD algorithm consists of two parts: the solution of the Poisson equation, and the explicit part which is based on stencil data structures to sweep the mesh and operate on scalar fields that represent physical variables. The first one can be considered as an independent algebraic module, while the latter is generally the most complex in terms of portability because it is tightly linked with the overall structure of the code. The effort of porting a source code is far from trivial, since it consists in rethinking and rewriting thousands of lines. The idea is to develop an implementation model that allows to separate the application logic from the particular system architecture. Another key point is that if an algorithm naturally fits stream processing, the

*Corresponding author. Email: cttc@cttc.upc.edu

most restrictive paradigm at the bottom level, then it will work well on upper levels (shared and distributed memory MIMD parallelization) and, it will work well on GPUs, MICs, and CPUs. This leads to a conclusion that a fully-portable algorithm must be composed only of operations that are SIMD-compatible.

Taking into account this diversity of frameworks and architectures, and the increasing complexity of programming models, the idea proposed in this paper is that the algorithm must: 1) only consist of operations compatible with stream processing (portability); 2) rely as much as possible on a minimal set of common linear algebra operations with standard interfaces (modularity); 3) the Poisson solver is considered as a black-box, linked with highly optimized libraries, or must be comprised by operations in agreement with the aforementioned points. This maximizes the independence of the implementation from a particular computing framework.

The algorithm for modeling of incompressible turbulent flows on unstructured meshes presented in this paper is based on only three linear operators: 1) the sparse matrix-vector product (SpMV); 2) the dot product; 3) the linear combination of vectors $\mathbf{y} = \alpha\mathbf{x}+\mathbf{y}$ (referred as AXPY in the BLAS standard nomenclature). This way, the problem of porting the code is reduced just to switching between existing implementations of these operations. The novelty of our approach relies upon the treatment of the non-linear operator of convection (and also diffusion in case LES is used). The proposed strategy consists in rewriting them as a concatenation of two sparse matrix-vector products.

For the sake of simplicity for the Poisson equation we use a preconditioned conjugate gradient (PCG) solver with the Jacobi preconditioner which fits the requirements of the algebraic operations. We can afford such a simple solver because the CFD algorithm uses an explicit time integration which implies a small time step. As a consequence, the pressure field and, respectively, the right-hand-side of Poisson equation doesn't change a lot within one time step. Therefore, using solutions from previous time steps as an initial guess helps notably to speedup the solution process.

It must be noted that the PCG with the SAI preconditioner only needs an SpMV operation on the solution stage, so it can be easily used in the proposed implementation approach. Furthermore, according to our experience (Oyarzun (14)), SAI can improve notably the solver performance especially in cases when more accurate solution of the Poisson equation is required.

Then, in order to prove its applicability, our portable approach has been implemented for hybrid supercomputers with GPU co-processors and standard multi-core systems. This has allowed us to carry out a detailed comparative performance analysis. This study aims at understanding the challenge of running CFD simulations on hybrid CPU/GPU supercomputers, and forming some fundamental rules to attain the maximal achievable performance. The Minotauro supercomputer of the Barcelona Supercomputing Center has been used for performance tests. The CUDA platform was chosen for implementing the main algebraic kernels. We use our in-house SpMV implementations optimized for the specific sparsity patterns that appear in the algorithm. The cuSPARSE library for CUDA also provides the necessary linear algebra operations. It was used as a reference for performance comparison and appeared to have slightly lower performance than the in-house SpMV versions (Oyarzun (14)). To ensure the portability of our kernels we have also made equivalent OpenCL implementations derived from CUDA in a very straightforward way and no significant changes in performance were observed.

Asynchronous communications overlapped with computations on accelerators are used for host-device and MPI data transfers. Scalability tests engaging up to 128 GPUs were performed and the results have been directly compared with executions on the same number of 6-cores CPUs. Finally, it is demonstrated that the overall performance of our CFD algorithm can be precisely estimated by analyzing only the parallel performance of the three basic algebraic kernels individually.

Regarding the related works, an early attempt of using GPU for CFD simulations is described in (Micikevicius (09)), where a CUDA implementation of a 3D finite-difference algorithm based on high-order schemes for structured meshes was proposed. Other examples of simulations on GPU using structured meshes can be found in (Alfonsi (11); Elsen (08)). However, these works focused only on a single-GPU implementation and rely on structured stencil data structures. OpenCL

implementations of such a class of algorithms for single GPU can be found, for instance, in (Rossi (13)), where a novel finite-element implementation based on edge stencils is presented, and in (Soukov (12)), where a set of basic CFD operations based on high-order schemes is studied.

Within the class of multi-GPU CFD implementations, a successful example of a high-order finite difference approach with a level set model for simulating two-phase flows can be found in (Jacobsen (13)). In addition, a Hybrid MPI-OpenMP-CUDA 3D solver is presented in (Zaspel (13)). Both implementations are restricted to structured grids. Some efforts for porting unstructured CFD codes to multi-GPU were conceived by porting only the most computational intensive parts of the algorithm (Poisson equation), as explained in (KhajehSaeed (13); Oyarzun (14)). Although, this methodology fails to attain the maximum of GPU potential because of Amdahl's law limitations. Finally, DNS simulation using unstructured meshes and multi-GPU platforms were shown in (Kampolis (10); Asouti (10)). The strategy adopted there was based on a vertex-centered finite volume approach including a mixed precision algorithm. Nevertheless, in all the mentioned examples the overall implementation seems to be tightly coupled with the framework it relies upon. Therefore, portability of those codes requires a complex procedure and large programming efforts. In contrast, the present paper focuses on a fully-portable implementation approach for a CFD algorithm, targeting any computing architecture or mesh type.

The rest of the paper is organized as follows: Section 2 describes the math background for the numerical simulation of incompressible turbulent flows; in Section 3 the portable implementation model based on algebraic operations is described; Section 4 focuses on its implementation on hybrid systems engaging both CPU and GPU devices; numerical experiments on the Minotauro supercomputer of the Barcelona Supercomputing Center are shown in Section 5; and, finally, concluding remarks are stated in Section 6.

2. Math model and numerical method

The simulation of a turbulent flow of an incompressible Newtonian fluid is considered. The flow field is governed by the incompressible Navier-Stokes equations written as:

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

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \nu \nabla^2 \mathbf{u} \tag{2}$$

where $\mathbf{u}$ is the three-dimensional velocity vector, p is the kinematic pressure scalar field and ν is the kinematic viscosity of the fluid.

In an operator-based formulation, the finite-volume spatial discretization of these equations reads

$$\mathbf{M}\mathbf{u}_c = \mathbf{0}_c, \tag{3}$$

$$\mathbf{\Omega}\frac{d\mathbf{u}_c}{dt} + \mathbf{C}(u_s)\mathbf{u}_c + \mathbf{D}\mathbf{u}_c + \mathbf{\Omega}\mathbf{G}p_c = \mathbf{0}_c, \tag{4}$$

where $\mathbf{u}_c$ and p_c are the cell-centered velocity and pressure fields, $\mathbf{0}_c$ is the discrete collocated field with zero in each component, u_s is the velocity field projected to the faces' normals, $\mathbf{\Omega}$ is a diagonal matrix with the sizes of control volumes, $\mathbf{C}(u_s)$ and $\mathbf{D}$ are the convection and diffusion operators, and finally, $\mathbf{M}$ and $\mathbf{G}$ are the divergence and gradient operators, respectively. In this paper, a second order symmetry-preserving and energy conserving discretization is adopted (Trias (14)): the convective operator is skew symmetric, $\mathbf{C}(\mathbf{u}_c) + \mathbf{C}(\mathbf{u}_c)^* = 0$, the diffusive operator is symmetric positive-definite and the integral of the gradient operator is minus the adjoint of the divergence operator, $\Omega\mathbf{G} = -\mathbf{M}^*$. Preserving the (skew-) symmetries of the continuous differential operators has shown to be a very suitable approach for accurate numerical simulations (Verstappen (03); Lehmkuhl (14); Rodriguez (11, 15)). For the temporal discretization, a second order explicit

Adams-Bashforth scheme is used, and the fully-discretized problem reads

$$\mathbf{\Omega}\frac{\mathbf{u}_c^{n+1}-\mathbf{u}_c^n}{\Delta t} = \mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right) + \mathbf{M}^* p_c^{n+1}, \quad (5)$$

$$\mathbf{M}\mathbf{u}_c^{n+1} = \mathbf{0}_c, \quad (6)$$

where $\mathbf{R}(\mathbf{u}_c) = -\mathbf{C}(u_s)\mathbf{u}_c - \mathbf{D}\mathbf{u}_c$. The pressure-velocity coupling is solved by means of a classical fractional step projection method (Chorin (68)). In short, reordering Eq. 5 is obtained the next expression for $\mathbf{u}_c^{n+1}$

$$\mathbf{u}_c^{n+1} = \mathbf{u}_c^n + \Delta t \mathbf{\Omega}^{-1}\left(\mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right) + \mathbf{M}^* p_c^{n+1}\right). \quad (7)$$

Then, substituting this into Eq. 6, leads to a Poisson equation for p_c^{n+1},

$$-\mathbf{M}\mathbf{\Omega}^{-1}\mathbf{M}^* p_c^{n+1} = \mathbf{M}\left(\frac{\mathbf{u}_c^n}{\Delta t} + \mathbf{\Omega}^{-1}\mathbf{R}\left(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}\right)\right), \quad (8)$$

which must be solved once per time-step. The left hand side of (8) is the discrete Laplace operator, $\mathbf{L} = -\mathbf{M}\mathbf{\Omega}^{-1}\mathbf{M}^*$, which is symmetric and negative definite.

At each time step, the non-linear convective operator is re-evaluated according to the velocity at the faces of the control volumes, $\mathbf{C}(u_s)$. In our collocated scheme the evaluation of the velocity at the faces is based on (Jofre (14)). Two additional operators are required: $\mathbf{\Gamma}_{c\to s}$ to project a cell-centered vector field to the faces' normals; and $\mathbf{G}_s$, to evaluate the gradient of a face-centered scalar field. The evaluation of u_s reads:

$$u_s = \mathbf{\Gamma}_{c\to s}\mathbf{u}_c^{n+1} - \Delta t\left(\mathbf{G}_s p_c^{n+1} - \mathbf{\Gamma}_{c\to s}\mathbf{g}_c^{n+1}\right) \quad (9)$$

where $\mathbf{g}_c^{n+1}$ is the cell-centered pressure gradient field.

In addition, when the LES model is activated, the viscosity at the faces ν_s needs to be updated at each time-step according to the turbulence eddie viscosity at faces ν_{ts}. As a result, the diffusive term becomes a non-linear operator that also needs to be re-evaluated at each time-step as $\mathbf{D}(\nu_s)$. The computation of $\nu_{ts} = \mathbf{K}(\mathbf{u}_c)$, where $\mathbf{K}$ denotes the contribution of a turbulence model, requires the calculation of the velocity gradients to construct the tensor operators, and, depending of the LES model, perform certain tensor operations. Various modern LES models fit this approach, including Smagorinsky, WALE, QR, Sigma, S3PQR. For instance, the WALE model used in the numerical experiments is defined as follows:

$$\nu_{ts} = (c_{wale} l)^2 \frac{(\overline{\mathcal{V}} : \overline{\mathcal{V}})^{3/2}}{(\overline{\mathcal{S}} : \overline{\mathcal{S}})^{5/2} + (\overline{\mathcal{V}} : \overline{\mathcal{V}})^{5/4}} \quad (10)$$

where c_{wale} is the model constant, here $c_{wale} = 0.325$, l is the length of the filter, $\overline{\mathcal{S}}$ is the filtered strain-of-rotation tensor, and $\overline{\mathcal{V}}$ is the traceless symmetric part of the square of the velocity gradient tensor. The viscosity is recalculated each iteration as $\nu_s^{n+1} = \nu_s + \nu_{ts}^{n+1}$, consequently the diffusion operator is updated (for details about models see, for instance, (Trias (15)) and references therein). Further details on this integration method and some options for the definition of the discrete operators can be found in (Trias (14); Jofre (14)). The overall time-step algorithm is outlined in Algorithm 1.

Algorithm 1 Time integration step

1: Evaluate the predictor velocities: $\mathbf{u}_c^p = \mathbf{u}_c^n + \Delta t \Omega^{-1} \left(\mathbf{R}(\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}) \right)$
($\mathbf{R}$ evaluated according to u_s and ν_s)
2: Solve the Poisson equation: $\mathbf{L}p_c^{n+1} = \frac{1}{\Delta t} \left(\mathbf{M}\mathbf{u}_c^p \right)$
3: Correct the centered velocities: $\mathbf{g}_c^{n+1} = \mathbf{G}p_c^{n+1}$, $\quad \mathbf{u}_c^{n+1} = \mathbf{u}_c^p - \Delta t(\mathbf{g}_c^{n+1})$
4: Calculate the velocities at the faces: $u_s = \mathbf{\Gamma}_{c\rightarrow s}\mathbf{u}_c^{n+1} - \Delta t \left(\mathbf{G}_s p_c^{n+1} - \mathbf{\Gamma}_{c \rightarrow s}\mathbf{g}_c^{n+1} \right)$
5: Calculate the eddie viscosity (if LES used): ${\nu_t}_s^{n+1} = \mathbf{K}(\mathbf{u}_c^{n+1})$, $\nu_s^{n+1} = \nu_s + {\nu_t}_s^{n+1}$
6: Calculate Δt based on the CFL condition: $\Delta t = CFL(\mathbf{u}_c^{n+1})$

3. Operational algebraic implementation approach

Leaving aside the linear solver, the most common form of implementing a CFD code is by using stencil data structures. This is how it is arranged our CFD code, TermoFluids (Lehmkuhl (07); Borrell (16)), that is an object-oriented code written in C++. TermoFluids includes a user-friendly API to manage the basic discrete operations of the geometric discretization. This API is used on the pre-processing stage to generate stencil raw data structures, storing in a compact form the geometric information required by the numerical methods. On the time integration phase, where most of computing time is spent, computations are based on sweeping through the stencil arrays and operating on scalar fields that represent physical variables. Using raw flattened stencil data structures, rather than higher-level object-oriented intuitive API, optimizes the memory usage and increases the arithmetic intensity of the code resulting in higher performance. In this paper we present a new implementation approach where stencil data structures and stencil sweeps are replaced by algebraic data structures (namely, sparse matrices) and purely algebraic kernels (SpMV). The high-level abstractions based on object-oriented programming are the same but the data structures at the bottom layer, used to accelerate the time integration phase, are sparse matrices stored in compressed formats instead of stencil arrays. Both implementations are equivalent in terms of the physical results and very similar in terms of performance. However, with the algebraic approach a perfect modularity is achieved since the code is mainly reduced to three algebraic kernels: the sparse-matrix vector product (SpMV), the linear combination of two vectors (denoted “AXPY” in the BLAS standard) and the dot product of two vectors (DOT). In the numerical experiments section it is shown how these three kernels represent more than 95% of the computing time. As a result, the portability of the code becomes also straightforward since we need to focus only on the three algebraic kernels.

The linear operators that remain constant during all the simulation can be evaluated as a sparse-matrix vector product in a natural way. For the non-linear operators, such as the convective term, the sparsity pattern remains constant during all the simulation but the matrix coefficients change. For these operators, we have followed the strategy of decomposing them into two SpMV: the first product is to update the coefficients of the operator, and the second to apply it.

In particular, the coefficients of the convective operator are updated at each time step according to u_s. If N_f is the number of mesh faces, u_s is a scalar field living in $\mathbb{R}^{N_f}$. On the other hand, in practice, the coefficients of the convective term are stored in a one-dimensional array of dimension N_e, where N_e is the number of non-zero entries in $\mathbf{C}(u_s)$. The arrangement of this array depends on the storage format chosen for the operator. Under these conditions, we define the evaluation of $\mathbf{C}(u_s)$ as a linear operator $\mathbf{E_C} : \mathbb{R}^{N_f} \mapsto \mathbb{R}^{N_e}$, such that:

$$\mathbf{C}(u_s) \equiv \mathbf{E_C} u_s. \tag{11}$$

Therefore, the evaluation of the non-linear term, $\mathbf{C}(u_s)\mathbf{u}_c$, results on the concatenation of two SpMV. In particular, the definition of $\mathbf{E_C}$ for the sliced ELLPACK (Monakov (10)) storage format used in this paper is presented in Section 4.

In an analogous way, when the LES model is activated, the coefficients of the diffusive term need

Step of Algorithm 1	SpMV	axpy	dot	extras
1 - predictor velocity	8	6	0	0
2.1 - Poisson equation (r.h.s)	3	1	0	0
2.2 - **Poisson equation (per iteration)**	**2**	**3**	**2**	**0**
3 - velocity correction	3	3	0	0
4 - velocity at faces	7	0	0	0
5 - eddie viscosity (optional)	9	0	2	1
6 - CFL condition	0	0	0	1
Total outside Poisson solver	**30**	**10**	**2**	**2**

Table 1. Number of times that each basic operation is performed in the numerical algorithm

to be updated according to ν_s. Therefore, the evaluation of the diffusive operator is performed as a linear map $\mathbf{E_D} : \mathbb{R}^{N_f} \mapsto \mathbb{R}^{N_e}$, such that:

$$\mathbf{D}(\nu_s) \equiv \mathbf{E_D}\nu_s. \tag{12}$$

This strategy allows the evaluation of the non-linear operators by calling two consecutive SpMV kernels with constant coefficients, without adding new functions to the implementation. Table 1 sums up the number of times that each kernel is called at the different steps of Algorithm 1. The column "extra" corresponds to operations different from our three main algebraic kernels. In Section 5 is shown that these operations have a relatively negligible computing cost.

In our implementation the vector fields $\mathbf{u}_c$ and $\mathbf{g}_c$ are stored as three scalar fields, one for each cartesian component. Therefore the linear operators applied to them result in three SpMV calls. In particular, for the convective and diffusive terms, the three components are multiplied by the same operator, so this can be optimized by using a generalized SpMV (see Section 4). On the other hand, the vectorial operator $\mathbf{G}$ is decomposed into the matrices $\mathbf{G}_x$, $\mathbf{G}_y$,$\mathbf{G}_z$ which are operated independently.

In the first step of Algorithm 1 the *SpMV* kernel is called eight times: one to re-evaluate the coefficients of the convective operator ($\mathbf{E_C}$), then (considering the LES model activated) another one is needed to update the diffusive operator ($\mathbf{E_D}$), and finally six calls are required to apply the convective ($\mathbf{C}$) and diffusive ($\mathbf{D}$) operators to the velocity components. Additionally six AXPY operations are performed, three to evaluate the linear combination of the velocities ($\frac{3}{2}\mathbf{u}_c^n - \frac{1}{2}\mathbf{u}_c^{n-1}$) and three more to multiply by Ω^{-1}, that is a diagonal matrix stored as a scalar field. Step 2 is separated into two sub-steps: firstly, the right hand side of the Poisson equation is calculated, here the divergence operator ($\mathbf{M}$) requires 3 SpMV; secondly, the preconditioned conjugate gradient (PCG) method is used to solve the Poisson equation. Within a PCG iteration one SpMV, three AXPY and two DOT are performed, the preconditioner (it can be a sparse approximate inverse, for instance, or the Jacobi diagonal scaling which is used in the present work) is counted as an additional SpMV. In the step 3 the velocity is corrected using the pressure gradient, the gradient operator ($\mathbf{G}$) requires three SpMV. The projection of the velocities at the faces in step 4 requires six SpMV coming from the operator $\mathbf{\Gamma}_{c\rightarrow s}$ and one instance of $\mathbf{G}_s$.

When LES model is activated, the eddie viscosity is evaluated ($\nu_{ts} = \mathbf{K}(\mathbf{u}_c)$) at the fifth step of Algorithm 1. The most costly part of this computation is the evaluation of a tensor dot product over the gradients of the velocity fields. For portability purposes, the linear part of the LES model, that derives on nine SpMV and two DOT, is separated from "extra" operations (`pow exp ,etc`) that depend on the model selected. Moreover, if a dynamic choice of the time step size is enabled one more extra operation is performed at step 6 . This operation consists on calculating the local CFL condition and obtaining the minimum value across the mesh cells.

In summary, the explicit part of the time step requires 30 calls to the SpMV kernel, 10 AXPY, 2 DOT products and few additional operations on the evaluation of the turbulence model and the

CFL condition. These "extra" operations are simple kernels compatible with stream processing and easily portable to any architecture. On the other hand, for the PCG solver: 2 SpMV, 2 DOT and 3 AXPY are required per iteration.

4. Implementation for a hybrid CPU/GPU supercomputer

Once the CFD algorithm has been reconstructed in a portable way, based on an algebraic operational approach, our aim is to implement this strategy to port our code to hybrid architectures engaging both multi-core CPUs and GPU accelerators. The introduction of accelerators into leading edge supercomputers has been motivated by the power constraints that require to increase the FLOPS per watt ratio of HPC systems. This seems to be a consolidated trend according to (Top500 (16)). Therefore, this development effort is aligned with the current HPC evolution trend (Oyarzun (17)). In particular, our computing tests were performed on the Minotauro supercomputer of the Barcelona Supercomputing Center (BSC). Its nodes are composed of two 6-core Intel Xeon E5649 CPUs and two NVIDIA M2090 GPUs, and are coupled via an InfiniBand QDR interconnect. The kernels were implemented in CUDA 5.0 since it is the natural platform to implement a code in NVIDIA GPUs (Nvidia (07)). Moreover the availability of the cuBLAS and cuSPARSE libraries provides all the necessary linear algebra operations, making the code portability straightforward. Nevertheless, for the SpMV kernel, we have focused on optimizing the implementation targeting our specific application context, *i.e.* targeting the sparsity pattern derived from our unstructured discretizations. On the other hand, note that the AXPY and DOT are algebraic kernels independent of any application context, so we have relied on the efficient implementations of the cuBLAS 5.0 library. The rest of this section is focused on the implementation of the SpMV kernel on single-GPU and multi-GPU platforms.

Theoretical performance estimation

For a performance estimation of the SpMV we count the floating point operations to be performed and the bytes to be moved from the main memory to the cache, then we can estimate the cost of both processes by comparing with the presumed computing performance and memory bandwidth of the device where the kernel is executed.

We consider as a representative problem the discrete Laplacian operator over a tetrahedral mesh, a second order discretization results in 5 entries per row (diagonal + 4 neighbors of a tetrahedron, except boundary cells). Therefore, if the mesh has N cells the Laplacian operator will contain approximately $5N$ entries. The size of the matrix in memory is $60N$ bytes (with double precision 8-byte values): $5N$ double entries ($40N$ bytes), plus $5N$ integer entries ($20N$ bytes) to store the column indices of the non-zero elements (additional elements to store row indices depend on the chosen storage format). We need to add also the two vectors engaged in the SpMV which contribute with $16N$ additional bytes. Regarding the arithmetics, 5 multiplications and 4 additions are required per each row of the Laplacian matrix, so this results in a total of $9N$ floating point operations. In our performance estimation we assume an infinitely fast cache, zero-latency memory and that each component of the input vector is read only once, i.e. ideal spatial and temporal locality. For the NVIDIA M2090 accelerator with ECC mode enabled the peak memory bandwidth is 141.6 GB/s. Therefore, the total time for moving data from DRAM to cache is $76N/141.6 = 0.54Nns$. On the other hand, the peak performance of the NVIDIA M2090 for double precision operations is 665.6 GFLOPS (fused multiplication addition is considered). Therefore, the total time to perform the floating point operations of the SpMV is estimated as: $9N/665.6 = 0.014Nns$, which is 38× lower than the time to move data from DRAM to cache. This is therefore a clearly memory-bounded kernel characterized by a very low FLOP per byte ratio: $9N/76N = 0.12$. Thus, the efficiency of the implementation basically depends on the memory transactions rather than on the speed of computations. Under these conditions, a tight upper bound for the achievable performance is the

product of the arithmetic intensity by the device bandwidth, this gives 16.8 GFLOPS, *i.e.* 2.5% of the peak performance of the M2090 GPU. With this estimation in mind, that is based in some optimistic assumptions, special attention has been paid to the memory access optimization.

Heterogeneous implementation

The utilization of all resources of a hybrid system, such as Minotauro supercomputer, to run a particular kernel requires a heterogeneous implementation. As shown in the previous subsection, the maximum performance achievable with the SpMV kernel is proportional to the memory bandwidth of the device where the kernel is executed. The memory bandwidth of the CPU and the GPU composing the Minotauro supercomputer is 32GB/s and 142 GB/s, respectively. Thus, in idealized conditions the GPU outperforms the CPU by 4.4×. Consequently, considering this ratio, a balanced workload distribution would be 82% of the rows for the GPU and 18% for the CPU. However, this partition requires a data-transfer process between both devices that generates and additional overhead. This leaves a margin of improvement that can only be profited by using complex heterogeneous implementations. Recent developments (Yang (17)), based on distributing the different parts of a hybrid storage format, have shown up to 11.07% time reductions. Another heterogenous algorithm is presented in (Yang (15); Li (2015)), their idea consists in determining the partitions using a probabilistic mass function to represent the distribution pattern of the non-zeros, and afterwards distribute them accordingly. By doing so, they reported an average improvement of 15.75% comparing to similar heterogeneous solutions. However in the parallel implementation (described latter in this section), CPUs are used only to asynchronously manage inter-GPU communications by overlapping them with calculations on the GPUs. We expect for the future a closer level of integration between CPUs and GPUs, given by improved interconnection technologies or, as shown in different examples, by the integration of both devices on a single chip.

Unknowns reordering

A two-level reordering is used in order to adapt the SpMV to the stream processing model and to improve the efficiency of the memory accesses. Firstly, the rows are sorted according to the number of non-zero elements. For hybrid meshes this means that the rows corresponding to tetrahedrons, pyramids or prisms are grouped continuously. The boundary elements are reordered to be at the end of the system matrix. This ordering aims to maintain regularity from the SIMD point of view: threads processing rows of the same group have identical flow of instructions. Secondly, a band-reduction Cuthill-McKee reordering algorithm (Cuthill (69)) is applied at each subgroup in order to reduce the number of cache misses when accessing the components of the multiplying vector. Note that in the SpMV kernel the random memory accesses affect only the multiplying vector, the matrix coefficients and the writes on the solution vector are sequential. Each component of the multiplying vector is accessed as many times as neighbors has the associated mesh cell, so this is the measure of the potential temporal locality or cache reuse. The unknowns reordering is a costly operation in comparison with SpMV, but it is performed only once at the preprocessing stage so it becomes negligible.

Generalized SpMV

Another way to reduce the memory communications is by grouping wherever possible different SpMV into a so-called "Generalized SpMV" (GSpMV), that multiplies the same matrix to multiple vectors simultaneously. For instance, the velocity vector in any cell is described by three components (u, v, w), which in practice are stored on three independent scalar fields. Some operators are multiplied by each of the three velocity components, resulting in three calls to the SpMV kernel using the same sparse matrix. Therefore, if the components are operated independently, the

same matrix coefficients and indices are fetched three times from the DRAM to the cache. With the GSpMV the matrix elements are fetched only once and the arithmetic intensity increases by $\approx 2.1\times$. However the reuse of components of the multiplying vector stored in the cache may reduce, because it is filled with elements of the three velocity vectors instead of only one. Nevertheless, in our numerical experiments we have obtained in average 30% time reduction by using the GSpMV.

Storage formats

Given a sparse matrix, its sparsity pattern determines the most appropriate storage format. The goal is to minimize the memory transmissions and increase the arithmetic intensity by: i) reducing the number of elements required to describe the sparse matrix structure, ii) minimizing the memory divergence caused by its potential irregularity. The sparse matrices used in our CFD algorithm arise from discretizations performed on unstructured meshes. Considering that we are using second order schemes for the discretization of the operators, and that the typical geometrical elements forming the mesh are tetrahedrons, pyramids, prisms and hexahedrons, generally the number of elements per row ranges between 5 and 7 for the interior elements and is 1 or 2 for the few rows associated to boundary nodes. Since we are not concerned in modifying the mesh, we restrict our attention to static formats, without considering elements insertion or deletion.

With this scenario in mind, an ELLPACK-based format is the best candidate to represent our matrices. The standard ELLPACK format is parametrized by two integers: the number of rows, N, and the maximum number of non-zero entries per row, K. All rows are zero-padded to length K forcing regularity. The matrix is stored in two arrays: one vector of doubles with the matrix coefficients (`Val`), and a vector of integers with the corresponding column indices (`Col`). This regularity, benefits the stream processing model because each equal-sized row can be operated by a single thread without imbalance. However, the zero-padding to the maximum initial row-length can

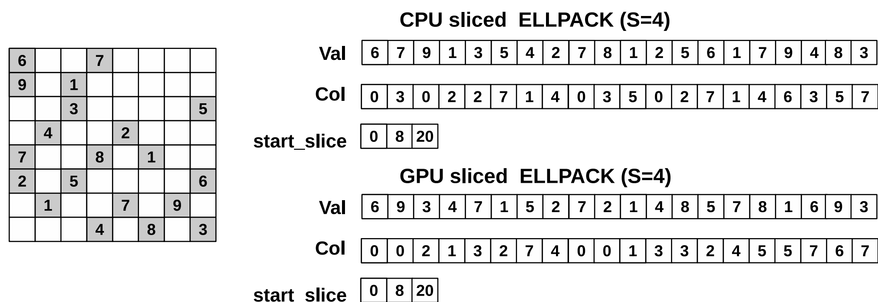

Figure 1. An example of sELL format and its memory layout for CPUs and GPUs.

produce an important sparse overhead, specially when there is a reduced number of polygons in the mesh with maximal number of faces. To overcome this overhead, a generalization of the previous algorithm, called sliced ELLPACK (sELL)has been implemented based on (Monakov (10)). In the sELL format the matrix is partitioned into slices of S adjacent rows, which are stored using the ELLPACK format. The benefit of this approach is that each slice has its particular K parameter, reducing the overall zero-padding overhead. The reordering strategy described above, i.e. grouping rows by its entries, improves the efficiency of the sELL format since minimizes the zero entries that need to be padded. In the sELL format it is added an integer-vector, called `slice_start`, holding the indices of the first element on each slice. Thus, for the i'th slice, the number of non-zero entries per row (including the zero-padded elements) can be calculated as

$$K_i = \frac{slice_start[i+1] - slice_start[i]}{S}.$$

If $S = N$ the sELL storage format becomes the standard ELLPACK. On the other hand, if $S = 1$

no zeros are padded and the format becomes equivalent to the Compressed Sparse Row format (CSR). For GPU implementations S is set equal to the number of threads launched per block, this way a single thread is associated to each row of the slice. The index of the row processed by each thread is calculated as `rowid = threadIdx.x + BlockIdx.x*BlockDim.x`. Zeros are padded in the case that a slice is composed by rows with different number of entries.

Figure 1 shows the storage of a sparse matrix using the sELL format for both a CPU and a GPU execution. In this case S=4 and the matrix is divided in two slices with 3 and 4 elements per row, respectively. However, note that the memory layout is different in both cases. In the CPU execution the elements are stored in row-major order within each slice, because a sequential process operates one row after the other. In the GPU execution a single thread is associated to each row, consequently, the elements are stored in column-major order to achieve coalesced memory accesses to the `Val` and `Col` arrays. Further details of this aspects can be found in (Bell (08)). Table 2 compares the entries required with the ELLPACK and sliced ELLPACK formats to store matrices coming from different discretizations. The sELL format requires around 16% less entries because less zero-padding is needed to achieve the desired regularity.

Rows	Pyramids	Tetrahedrons	total entries ELL	total entries sELL	Reduction
$500,000$	3.80%	96.30%	$3,000,000$	$2,519,000$	16.03%
$1,500,000$	2.62%	97.38%	$9,000,000$	$7,539,300$	16.23%
$5,500,000$	1.45%	98.55%	$33,000,000$	$27,579,750$	16.43%

Table 2. Entries required to store our discretization matrices with ELL and sELL formats

Non linear operators

Since the sliced ELLPACK storage format has already been described in previous subsection, now it can be explicitly defined the respective operator $\mathbf{E_C} : R^{N_f} \mapsto R^{N_e}$ that, given u_s, generates the array `Val` storing the corresponding coefficients of the convective operator in the sELL format. However, first of all the discretization of the convective operator needs to be explicitly specified:

$$[\mathbf{C}(u_s)\mathbf{u}_c]_i = \sum_{j\in nb(i)} (\mathbf{u}_{cj} + \mathbf{u}_{ci}) \frac{u_{sij} A_{ij}}{2} \quad (13)$$

where i is a mesh node, $nb(i)$ are its neighboring nodes, A_{ij} is the area of the face between node i and node j, and u_{sij} the corresponding component of the field u_s. For each node i, the indices of the non-zero entries of its row are: $entries(i) = nb(i) \cup i$. Being these indices sorted in ascending order, we refer to the relative position of any index $j \in entries(i)$ with the notation $ord_i(j)$. For each ordered pair (i, j) of neighboring nodes, the next two coefficients are introduced to $\mathbf{E_C}$:

$$\mathbf{E_C}(ind(i,j), ij) = \mathbf{E_C}(ind(i,i), ij) = \frac{A_{ij}}{2} \quad (14)$$

where ij is the index corresponding to the face between nodes i and j, and the function $ind(i, j)$, that fixes the layout of the elements of the convective operator in the array `Val`, is defined as:

$$ind(i,j) = i\%S + ord_i(j) * S + \sum_{l<\frac{i}{S}} K_l * S, \quad (15)$$

where S and K_l are parameters of the sELL format described in previous subsection. The first two terms of Equation 15 define the position of the new index within its slice: the first term represents

the row and the second the column. The third term is the offset corresponding to all previous slices.

The sparse matrix $\mathbf{E_C}$ has constant coefficients and is also stored in a compressed format. However, its rows can not be reordered since the order is determined by the sELL layout of $\mathbf{C}(u_s)$. Therefore, with the purpose of avoiding a high sparse overhead produced by the zero-padding, $\mathbf{E_C}$ is stored using the standard CSR format (Bell (08)).

Finally, note that the definition of $\mathbf{E_D}$, the sparse linear operator used to evaluate the coefficients of $\mathbf{D}(\nu_s)$, is performed following the same strategy than with $\mathbf{E_C}$.

Multi-GPU parallelization

At the top level the parallelization strategy to run the code on multiple GPUs is based on a standard domain decomposition: the initial mesh, $\mathcal{M}$, is divided into P non-overlapping sub-meshes $\mathcal{M}_0, ..., \mathcal{M}_{P-1}$, and an MPI process executes the code at each sub-mesh. For the $i'th$ MPI process its unknowns can be categorized into different sub-sets:

- *owned* unknowns are those associated to the nodes of $\mathcal{M}_i$;
- *external* unknowns are those associated to the nodes of other sub-meshes;
- *inner* unknowns are the owned unknowns that are coupled only with other owned unknowns;
- *interface* unknowns are the owned unknowns coupled with external unknowns;
- *halo* unknowns are the external unknowns coupled with owned unknowns.

In our MPI+CUDA implementation, the mesh is divided into as many sub-domains as GPUs engaged. The communication episodes are performed on three stages: i) copy data from GPU to CPU, ii) perform the required MPI communication, iii) copy data from CPU to GPU. Note that the inter-CPU communications are performed through the system network while the communications between the CPU and GPU through the PCI-e bus. For the SpMV kernel, we have developed a classical overlapping strategy, where the sub-matrix corresponding to the inner unknowns is multiplied at the same time that the halo unknowns are updated. Once the updated values of the halo unknowns are available at the GPU, the sub-matrix corresponding to the interface unknowns can also be multiplied. An schematic representation of this two-stream concurrent execution model is depicted in Figure 2. Note that a synchronization episode is necessary after both parts of the matrix are multiplied. This strategy is really effective because communications and computations are performed simultaneously on independent devices that do not disturb each-other (see examples on Section 5).

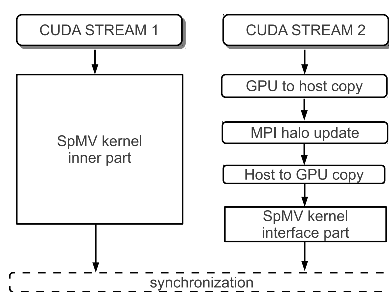

Figure 2. Two-stream concurrent execution model.

In order to facilitate and optimize the partition of the sparse matrices to perform the overlapped parallel SpMV, a first reordering of variables is performed forming three groups: inner, interface and halo variables. This ordering precedes and is prioritized versus the orderings described above for grouping rows with the same number of elements and for band-reduction. Therefore, for parallel executions a three-level reordering strategy is applied in order to optimize the data locality and

regularity. Further details of our overlapping strategy can be found in our previous work (Oyarzun (14)).

5. Computing experiments and performance analysis

The performance testing has been carried out on the Minotauro supercomputer of the Barcelona Supercomputing Center (BSC), its characteristics are summarized in Section 4. In the Minotauro supercomputer there is one 6-core CPU for each GPU, therefore, at comparing the multi-CPU vs the multi-GPU implementation we have respected this ratio running 6 CPU-cores for each GPU.

5.1. *Profiling the algorithm for a turbulent flow around a complex geometry*

First of all, the MPI-only and the MPI+CUDA versions of the code are profiled. The percentage of the total execution time spent in each algebraic kernel has been measured during the time integration process. The test case is the turbulent flow around the ASMO car geometry ($Re = 7 \times 10^5$), for which a detailed study was previously published in (Aljure (14)). The mesh has around 5.5 millions of control volumes that include tetrahedrons, pyramids and triangular prisms that are used in the boundary layer. The stopping criterion for the Poisson solver is set equal to 10^{-5} (the norm of the residual relative to the right-hand-side norm). In this particular case the average number of PCG solver iterations is 37 when using a Jacobi preconditioner. Additionally, the LES turbulence modeling is enabled with the WALE (Nicoud (99)) eddie-viscosity model.

The profiling results on different number of CPUs and GPUs are shown in Figure 3. Results for the multi CPU implementation show that the contribution of the SpMV in the overall algorithm stays constant at around 78% with a growth of the number of 6-core CPUs from 4 to 128. This indicates that the SpMV accelerates equally as the overall time-step. On the other hand, the AXPY operation is the leader in speedup: its contribution reduces from 14% to around 6%. This was expected since there are no communications in the AXPY operation. Finally, the contribution of the DOT operation grows with number of CPUs due to the collective reduction communications, its negative contribution is counteracted by the AXPY super-linearity. Finally, the remaining 2% of time is spent in other operations (the evaluation of the eddie viscosity on step 5 of Algorithm 1, and the evaluation of the CFL condition on step 6 of Algorithm 1). Therefore, the three basic algebraic kernels sum up to 98% of the time-step execution time on CPUs. This fact emphasizes the portability of the algorithm implementation.

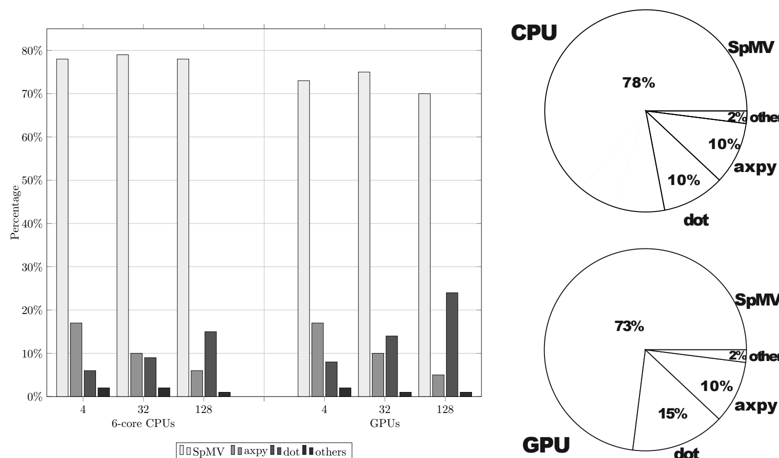

Figure 3. Relative weight of the main operations for different number of CPUs and GPUs (left) and diagrams of the average relative weight (right)

The GPU execution shows a lower percentage of time spent on the SpMV kernel. As demonstrated

in next subsection, this is due to a better exploitation of the device bandwidth by the SpMV kernel on GPUs. It is also observed a significant increase of the DOT percentage, which is mainly penalized by the all-reduce communication and the overhead of the PCI-Express host-device transactions.

In Figure 3 (right) are depicted the average results of the previous tests for different number of devices. Note that the distribution of the computing costs among the main kernels depends on the number of iterations required by the Poisson solver, because the distribution is substantially different for the Poisson solver and for the explicit part of the time step. Table 3 shows the relative weight of the main operations for these two parts separately. Results are similar for the GPU and CPU executions: the Poisson solver represents around 55% of the time-step and within it the SpMV kernel represents $57-67\%$ of the computing costs. One the other hand, in the explicit part the SpMV is more dominant, it represents more than 90% of the computing costs.

Device	Part of the code	Algebraic Operations			
		SpMV	AXPY	DOT	others
CPU	Poisson solver (55.81%)	**67.58%**	14.84%	16.97%	0.61%
	Explicit(44.19%)	**91.94%**	3.34%	1.66%	3.06%
	Time-step	**78.33%**	9.73%	10.24%	1.70%
GPU	Poisson solver (55.42%)	**57.71%**	17.61%	24.19%	0.49%
	Explicit (44.58%)	**91.67%**	2.10%	3.74%	2.49%
	Time-step	**72.53%**	10.45%	15.14%	1.88%

Table 3. Relative weight of the main operations for the Poisson solver, the explicit part and the overall time-step. Average of the results shown in Fig 3 for different number of devices

5.2. *Performance of SpMV on a single GPU and a single 6-core CPU*

The SpMV tests have been carried out for the discrete Laplace operator since it represents the dominant sparsity pattern. The Laplace operator has been discretized on 3D unstructured meshes. The number of non-zero entries per row ranges from 5 to 7 for tetrahedral, prismatic, pyramidal and hexahedral cells. Our implementation of the CPU and GPU SpMV with the sELL storage format is compared with general-purpose SpMV implementations of commonly used standard libraries: Intel Math kernel library (MKL) 13.3 for CPUs and cuSPARSE 5.0 for GPUs. Intel MKL only supports the CSR format, while cuSPARSE supports the CSR format and a hybrid (HYB) format, which automatically determines the regular parts of the matrix that can be represented by ELLPACK while the remaining are solved by COO format (Bell (08)). The execution on the 6-core CPUs employs OpenMP shared memory loop-based parallelization with a static scheduling. In all cases, the two-level reordering explained in Section 4 (grouping of rows with equal number of non-zero entries + band reduction reordering) is used in order to improve the memory performance. Unknowns reordering results in 40% time reduction in average for CPU executions.

The achieved net performance, in GFLOPS for the different storage formats under consideration and for different mesh sizes is shown in Table 4. In all the cases our in-house sELL format shows the best performance. In average the speedup versus the Intel MKL and NVIDIA cuSPARSE library is 38% and 11%, respectively. Consequently, the sELL format has been chosen for the rest of this paper.

Device, SpMV format	Mesh size, thousands of cells					
	50	100	200	400	800	1600
CPU CSR *MKL*	2.45	2.18	1.49	1.37	1.30	1.18
CPU sliced ELLPACK	**3.44**	**3.02**	**2.89**	**2.76**	**2.41**	**2.06**
GPU CSR *cuSPARSE*	3.64	4.10	4.40	4.58	4.79	4.70
GPU HYB *cuSPARSE*	8.74	11.25	13.36	14.93	15.62	15.94
GPU sliced ELLPACK	**10.91**	**12.79**	**14.90**	**15.92**	**16.15**	**16.37**

Table 4. Net performance in GFLOPS obtained with different implementations of various matrix storage formats

Figure 4, shows the performance evolution of the sELL executions for matrices of different sizes and for both the CPU and GPU devices. There is a solid horizontal line on the plots that indicates an

estimation of the maximal theoretically achievable performance evaluated as explained in Section 4. Up to 90% of the peak estimation is achieved on the CPU and up to 97% on the GPU. It is important to note that the CPU shows better performance on a smaller matrices while the GPU on bigger ones. This trend can be clearly seen in Figure 4. The CPU performance decreases with the matrix size due to the increasing weight of cache misses. In other words, the perfect temporal locality assumed in our estimation is not met when the problem size grows. In particular, some of the multiplying vector components that need to be reused can not be held on the cache because of its limited size. On the GPU the effect is the opposite, there is a net performance growth with the matrix size due to the resulting higher occupancy of the stream multiprocessors, which allows to more efficiently hide memory latency overheads by means of hardware multi-threading. Saying it from the opposite perspective, if the amount of parallel threads (i.e. rows) is not enough the device can not be fully exploited. For the executions on the GPU the grid of threads was parametrized in accordance to the occupancy calculator provided by NVIDIA (Nvidia (07)). Moreover, on the NVIDIA M2090 GPU the distribution of the local memory between shared memory and L1 cache can be tuned. For the SpMV kernel the best performance is obtained when the maximum possible fraction of the local memory of the stream multiprocessors is used for cache functions (48KB).

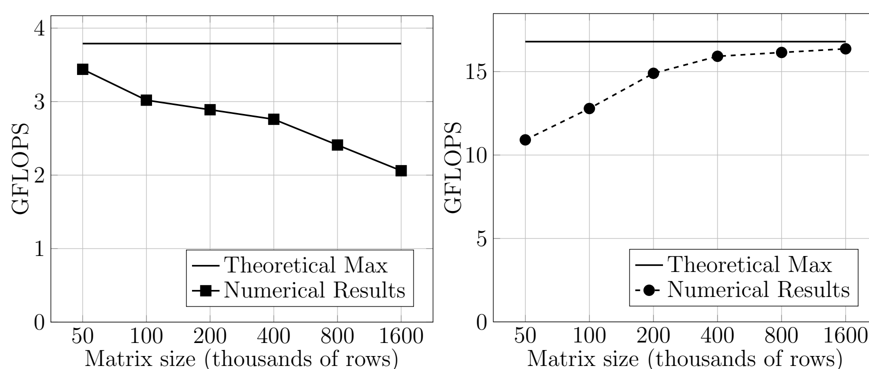

Figure 4. Net performance achieved on a single Intel Xeon E5649 6-core CPU (left) and a single NVIDIA M2090 GPU (right) for different mesh sizes. Solid line indicates an estimation of maximal theoretically achievable performance

Finally, Figure 5 depicts the speedup of the execution on a single GPU versus the execution on a single 6-core CPU. Due to the trends observed in the above-mentioned tests the speedup significantly grows with the mesh size, since the CPU performance goes down with size and the GPU performance goes up. The speedup ranges in the tests from around 3× up to 8×. Being the ratio between the GPU and CPU peaks memory bandwidth 4.4 (141.6GB/s for the GPU and 32GB/s for the CPU), and recalling that the SpMV is a clearly memory bounded operation, we can conclude from the result shown in Figure 5 that the bandwidth is better harnessed on the GPU than on the CPU, because in most of cases the speedup achieved is greater than 4.4. For the matrix with $50K$ rows the opposite result is observed, however this case is particularly small for devices with 6 GB and a 12 GB of main memory such as the the GPU and CPU under consideration, so it does not represent realistic usage.

5.3. *Parallel performance of the SpMV on multiple GPUs*

Firstly, we evaluate the benefit of overlapping computations with communications. In Figure 6 (left) is compared the time needed to perform the SpMV with and without overlapping, engaging 128 GPU for different matrix sizes. The matrices used range between 50 and 200 million (M) rows, corresponding to workloads between 0.4M and 1.6M per GPU, respectively. From 64% up to 86% of the communication costs are hidden behind computations using the overlapping strategy. The benefit increases with the problem size since the weight of the computations grows and the communications can be more efficiently hidden.

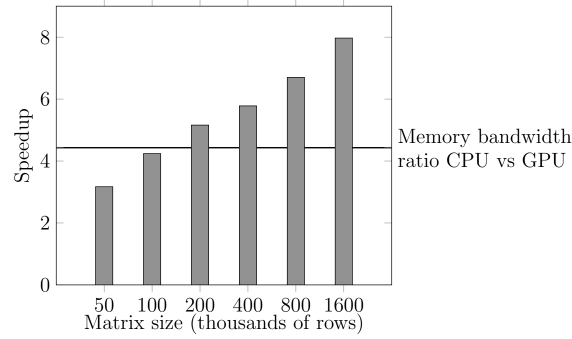

Figure 5. Speedup of the SpMV execution on a single GPU vs. on a 6-core CPU, for different mesh sizes.

All the remaining parallel performance tests are carried out using the overlap mode. Further details on the implementation of the SpMV with the overlapping strategy can be found in Section 4. Weak speedup tests for the SpMV on multiple GPUs are shown in Figure 6 (right). Three different local matrix sizes are considered: 0.4M, 0.8M, and 1.6M rows. The execution time grows for the biggest load only 1.2× while the problem size is increased 128×. For the smallest load considered this factor is around 2×. The slowdown is caused by an increase of the communication costs with the number of GPUs engaged, and is proportional to the relative cost of communications. The maximal load of 1.6M cells per GPU was chosen according to the limitation of the mesh size that could fit in memory if we run a complete CFD simulation. Furthermore, the strong speedup of the SpMV

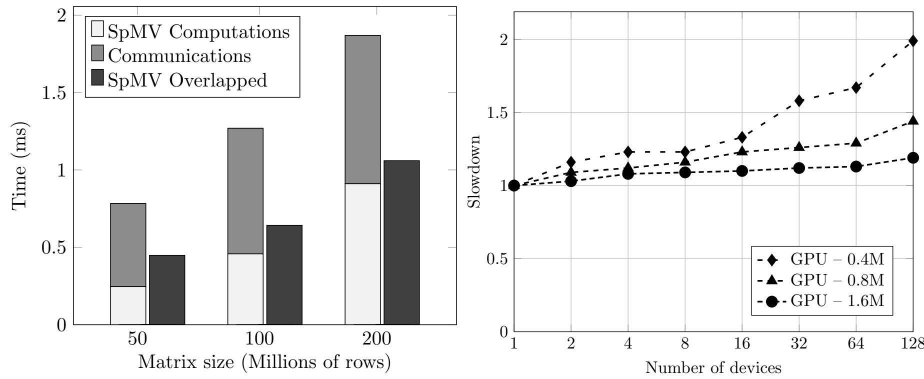

Figure 6. Left: Effect of overlapping communications and computations in the SpMV for different mesh sizes on 128 GPU. Right: Weak speedup tests for different local matrix sizes

kernel is shown in Figure 7 (left) for matrices of different sizes. Certainly, the larger is the size of the problem the better is the speedup obtained, because the relative weight of the communications is inversely proportional to the problem size. In the right part of the figure the same results are shown in terms of parallel efficiency (PE). It can be observed that in the range of tests performed the PE achieved mainly depends on the local workload rather than on the number of devices engaged: $\sim 80\%$ PE is obtained for a workload of 400K rows, $55 - 60\%$ for 200K rows, $37 - 41\%$ for 100K rows and $\sim 30\%$ for 50K rows. Therefore, if we are seeking for high PE we should not reduce the local problem below 400K rows per GPU. This result is consistent with the decrease of the GPU performance for matrices below 400K rows shown in Figure 4, that is produced by a lack of occupancy of the device. Indeed, note that the load of 400K unknowns could be considered a moderate one for a GPU with 6 GB of main memory. In any case, the resources engaged in a simulation should be consistent with this type of evaluations in order to use efficiently the granted computing time.

Figure 7 includes the strong speedup of the SpMV on up to 128 devices for the 12.8M cells mesh. The speedup on CPUs, 127×, is much higher than on GPUs, 53×. The PE degradation with the number of GPUs is twofold: i) an increase of the communications overhead; ii) the net performance

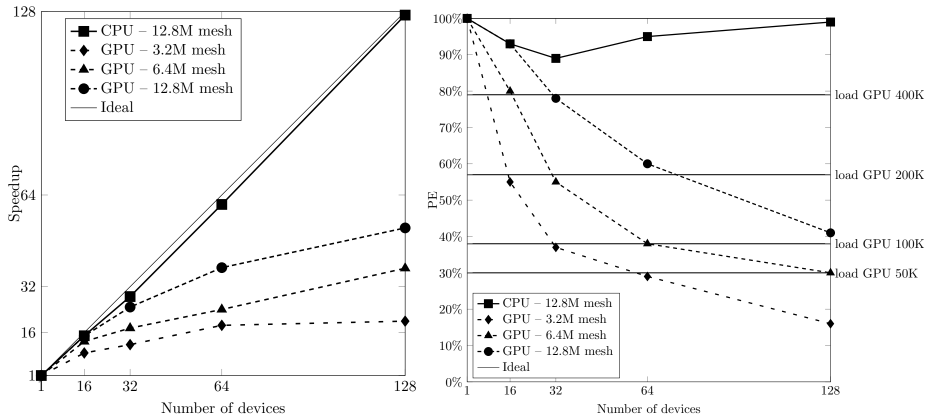

Figure 7. Strong speedup on multiple GPUs for different mesh sizes (left). Same result expressed in terms of parallel efficiency (right)

of each GPU reduces with local computing load due to occupancy fall. An opposite situation is observed for the CPUs: the performance increases when the local problem reduces due to better cache usage. Therefore, the increase of the communication overhead on CPU is compensated by the boost in the computing performance.

Figure 8 (left) demonstrates this effect in detail. It shows how the net performance of CPU and GPU changes when the number of devices increases (and the local problem size on each device reduces, respectively). Results are normalized by the performance achieved having the whole matrix on a single device. It can be observed that the CPU net performance increases more than twice while GPU performance goes down on 25%.

Figure 8 (right) shows an emulation of how the speedup plot would look like if the CPU and GPU performance (GFLOPS) would remain the same as the achieved on a single device, i.e. canceling cache and occupancy effects. We have “eliminated” both effects by considering a linear reduction of the computing time while keeping the same communication costs measured on the real tests. The CPU and GPU plots in this idealized case appear very close to each other, therefore, we conclude that eventually the speedup on the CPUs is produced by the cache driven super-linear acceleration of the local computations that lacks on the GPU. The variation between the real performance and the imaginary emulation is minimal on the GPUs.

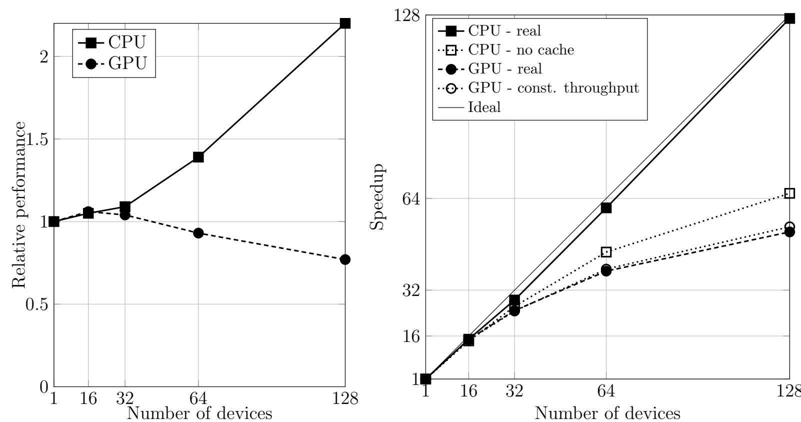

Figure 8. Relative net performance of CPUs and GPUs compared to the performance on a single device for the mesh of 12.8M cells (left); idealized emulation of how speedup would look like if the CPU and GPU performance would remain the same as on a single device (right)

The reason is that the communications dominate in the overlapping scheme, so canceling the degradation produced by the occupancy fall only affects the cost of the interface sub-matrix product (see Section 4). This result also demonstrates the efficiency of the overlapping communications scheme: CPU and GPU plots are very near, despite initially the GPU execution is $9\times$ faster.

5.4. *Experiments with complete CFD simulations*

The overall CFD algorithm has been evaluated on two test cases: the LES of the ASMO car ($Re = 7 \times 10^5$) executed on a mesh of 5.5M cells, that has been previously used for profiling; and the LES of a driven cavity ($Re = 10^5$) with a mesh of 12.8M cells (the driven cavity was chosen in this paper just for its simple geometry that makes generation of meshes of different sizes very easy). Figure 9 shows the solution time for different numbers of devices for both cases, we can derive from it the acceleration achieved and the difference between CPU and GPU executions. The dotted line represents the ideal scaling for each device with respect to the first execution (8 devices). The Poisson solver stopping criterion was set equal to 10^{-5} (the norm of the residual relative to the right-hand-side norm), and the LES turbulence modeling is enabled with the WALE (Nicoud (99)) eddie-viscosity model. It also includes estimations of the execution time. These are based only on time measurements for the three basic algebraic kernels, and the number of repetitions of each kernel per time-step (shown in Table 1). This simple formula reads:

$$
\begin{aligned}
T_{exp} &= 30T_{spmv} + 10T_{axpy} + 2T_{dot} \\
T_{imp} &= 2T_{spmv} + 3T_{axpy} + 2T_{dot} \\
T_{total} &= T_{exp} + it \cdot T_{imp}
\end{aligned}
$$

where the subindex of T indicates the corresponding kernel, and it refers to the iterations required by the Poisson solver.

Figure 9 shows, firstly, that the scalability on CPUs is better than on GPUs. Secondly, the performance on GPUs is always higher than on CPUs. For the ASMO car case the speedup of the GPU execution versus the CPU execution ranges from 6.3×, down to 2.6×, while for the DC case from 7.8× to 3.7×. Moreover, the estimation appears to be in good agreement with the real measurements, providing great portability in terms of performance evaluation: we can predict the performance and scalability of our code in any architecture by just studying the three main kernels separately.

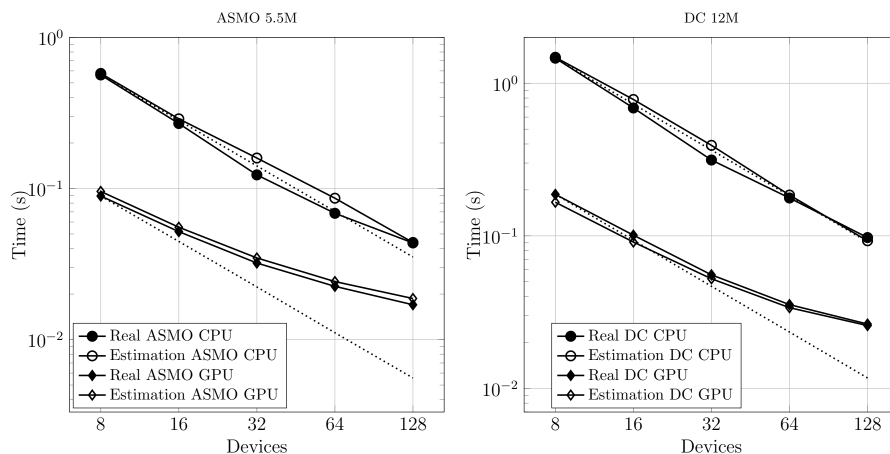

Figure 9. Strong speedup for the overall time-step (real and estimation), for the ASMO car with a mesh of 5.5M cells (left) and the Driven Cavity with a mesh of 12.8M cells (right)

6. Concluding remarks

The contribution of this paper is twofold. Firstly, we propose a portable modeling for LES of incompressible turbulent flows based on an algebraic operational approach. The main idea is substituting

stencil data structures and kernels by algebraic storage formats and operators. The result is that around 98% of computations are based on the composition of three basic algebraic kernels: SpMV, AXPY and DOT, providing a high level of modularity and a natural portability to the code. Among the three algebraic kernels the SpMV is the dominant one, it substitutes the stencil iterations. The non-linear terms, such as the convective operator, are also rewritten as two consecutive SpMVs. The modularity and portability requirements are motivated by the current disruptive technological moment, with a high level of heterogeneity and many computing models being under consideration without any clarity of which one is going to prevail.

The second objective has been the implementation of our model to run on heterogeneous clusters engaging both CPU and GPU co-processors. This objective is motivated by the increasing presence of accelerators on HPC systems, driven by power efficiency requirements. We have analyzed in detail the implementation of the SpMV kernel on GPUs, taking into account the characteristics of the matrices derived from our discretization. Our in-house implementation based on a sliced ELLPACK format clearly outperforms other general purpose libraries such as MKL on CPUs and cuSPARSE on GPUs. Moreover, for multi-GPU executions we have developed a communication-computations overlapping strategy. On the other hand, the AXPY and DOT kernels do not require specific optimizations because they are application-independent kernels with optimal implementations in libraries such as cuSPARSE.

Finally, several numerical experiments have been performed on the Minotauro supercomputer of the Barcelona Supercomputing Center, in order to understand in detail the performance of our code on multi-GPU platforms, and compare it with multi-core executions. First we have profiled the code for both implementations showing that certainly 98% of time is spent on the three main algebraic kernels. Then, we have focused on the SpMV kernel, we have shown its memory bounded nature, and how the throughput oriented approach of the GPU architecture better harnesses the bandwidth than the standard latency reducing strategy implemented on CPUs by means of caches and prefetching modules. The result is that although the bandwidth ratio between both devices is 4.4 the speedup of the GPU vs the CPU implementation reaches up to $8\times$ in our tests. Then the benefits of the overlapping strategy for multi-GPU executions has been tested, showing that large part of the communication (86%) can be hidden. We have also included strong and week speedup tests engaging up to 128 GPUs, in general good PE is achieved if the workload per GPU is kept reasonable. Considering the overall time-step, the multi-GPU implementation outperforms the multi-CPU one by a factor ranging between $3\times$ and $8\times$ depending on the local problem size. Finally, we have demonstrated that the performance of our code can be very well estimated by only analyzing the three kernels separately.

Acknowledgments

This work has been financially supported by the Ministerio de Ciencia e Innovación, Spain (ENE-2014-60577-R), the Russian Science Foundation, project 15-11-30039, CONICYT Becas Chile Doctorado 2012, the Juan de la Cierva posdoctoral grant (IJCI-2014-21034) and the Initial Training Network SEDITRANS (GA number: 607394), implemented within the 7th Framework Programme of the European Commission under call FP7-PEOPLE-2013-ITN. Our calculations have been performed on the Minotauro supercomputer at the Barcelona Supercomputing Center. The authors thankfully acknowledge these institutions.

References

G. Oyarzun, R. Borrell, A. Gorobets, A. Oliva, MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner, Computers & Fluids, Volume 92, 20 March 2014, Pages 244-252

Micikevicius P., 3D Finite Difference Computation on GPUs using CUDA, in: Proceeding of 2nd Workshop on General Purpose Processing on Graphics Processing Units,2009.

Alfonsi A, Ciliberti S., Mancini M, Primavera L. Performances of Navier-Stokes Solver on a Hybrid CPU/GPU Computing System. Parallel Computing Technologies. Lecture Notes in Computer Science Volume 6873, 2011, pp 404-416 (3D)

Elsen, E., LeGresley, P., Darve, E. Large calculation of the flow over a hypersonic vehicle using a GPU (2008) Journal of Computational Physics, 227 (24), pp. 10148-10161 (2D)

R. Rossi and F. Mossaiby and S.R. Idelsohn, A portable OpenCL-based unstructured edge-based finite element Navier-Stokes solver on graphics hardware, Computers & Fluids, Volume 81, 2013, Pages 134–144

S.A. Soukov, A.V. Gorobets, P.B. Bogdanov, OpenCL Implementation of Basic Operations for a High-order Finite-volume Polynomial Scheme on Unstructured Hybrid Meshes, Procedia Engineering, Volume 61, 2013, Pages 76–80

Dana A. Jacobsen, Inanc Senocak, Multi-level parallelism for incompressible flow computations on GPU clusters, Parallel Computing, Volume 39, Issue 1, 2013, Pages 1–20

Peter Zaspel, Michael Griebel,Solving incompressible two-phase flows on multi-GPU clusters,Computers & Fluids, Volume 80, 10 July 2013, Pages 356-364

Ali KhajehSaeed, J. Blair Perot , Direct numerical simulation of turbulence using GPU accelerated supercomputers,Journal of Computational Physics, vol. 235 (2013), pp.241-257.

I.C. Kampolis, X.S. Trompoukis, V.G. Asouti and K.C. Giannakoglou, CFD-based analysis and two-level aerodynamic optimization on graphics processing units, Computer Methods in Applied Mechanics and Engineering, vol. 199 (2010), pp. 712-722

V. G. Asouti, X. S. Trompoukis, I. C. Kampolis and K. C. Giannakoglou, Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units, International Journal for numerical methods in fluids, vol. 67 (2010), pp. 232-246.

F.X. Trias, O. Lehmkuhl, A. Oliva, C.D. Pérez-Segarra, R.W.C.P. Verstappen, Symmetry-preserving discretization of Navier–Stokes equations on collocated unstructured grids, Journal of Computational Physics, Volume 258, 1 February 2014, Pages 246-267, ISSN 0021-9991

R. W. C. P. Verstappen and A. E. P. Veldman. 2003. Symmetry-preserving discretization of turbulent flow. Journal of Computational Physics 187, 1 (May 2003), 343-368.

O. Lehmkuhl, I. Rodríguez, R. Borrell, J. Chiva and A. Oliva. Unsteady forces on a circular cylinder at critical Reynolds numbers, Physics of Fluids, 26, 125110 (2014)

I. Rodriguez, R. Borrell, O. Lehmkuhl, C.D. Pérez-Segarra and A. Oliva. Direct numerical simulation of the flow over a sphere at Re = 3700. Journal of fluid mechanics, May 2011, vol. 679, num. 2011, p. 263–287.

I. Rodriguez, O. Lehmkuhl, J. Chiva, R. Borrell, A. Oliva, On the flow past a circular cylinder from critical to super-critical Reynolds numbers: Wake topology and vortex shedding, In International Journal of Heat and Fluid Flow, Volume 55, 2015, Pages 91-103, ISSN 0142-727X

Chorin, A. J. Numerical solution of the Navier-Stokes equations Math. Comp. 22 (1968) pp. 745-762

L. Jofre, O. Lehmkuhl, J. Ventosa, F.X. Trias, A. Oliva .Conservation Properties of Unstructured Finite-Volume Mesh Schemes for the Navier-Stokes Equations. Numerical Heat Transfer, Part B: Fundamentals.Vol. 65, Iss. 1, 2014

F.X. Trias, D. Folch, A. Gorobets, A. Oliva. Building proper invariants for eddy-viscosity subgrid-scale models, Physics of Fluids, 27, 065103 (2015)

Lehmkuhl, O. and Perez-Segarra, C.D. and Borrell, R. and Soria, M. and Oliva, A. TermoFluids: A new Parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC Cluster. 2007. Lecture Notes in Computational Science and Engineering. Parallel Computational Fluid Dynamics 2007. Vol 67.pp 275-282.

R. Borrell, J. Chiva, O. Lehmkuhl, G. Oyarzun, I. Rodríguez, and A. Oliva. Optimising the Termofluids CFD code for petascale simulations. International Journal of Computational Fluid Dynamics Vol. 30 , Iss. 6,2016

A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures, High Performance Embedded Architectures and Compilers Lecture Notes in Computer Science Volume 5952, 2010, pp 111-125

Top500, Ranking of supercomputers according to LINPACK benchmark. http://www.top500.org

G. Oyarzun, R. Borrell, A. Gorobets, F. Mantovani, A. Oliva, Efficient CFD code implementation for the ARM-based Mont-Blanc architecture, Future Generation Computer Systems, Available online 20 Septem-

ber 2017, ISSN 0167-739X, https://doi.org/10.1016/j.future.2017.09.029.

Nvidia, Nvidia CUSPARSE and CUBLAS libraries (2007). http://developer.nvidia.com/cuda-toolkit

W. Yang, K. Li, K. Li, A hybrid computing method of SpMV on CPU-GPU heterogeneous computing systems, J. Parallel Distrib. Comput. (2017)

W. Yang, K. Li, Z. Mo and K. Li, Performance Optimization Using Partitioned SpMV on GPUs and Multicore CPUs, in IEEE Transactions on Computers, vol. 64, no. 9, pp. 2623-2636, Sept. 1 2015.

K. Li, W. Yang and K. Li, Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling, in IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 1, pp. 196-205, Jan. 2015.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices In Proc. 24th Nat. Conf. ACM, pages 157–172, 1969.

N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, Nvidia Technical Report NVR-2008-004, Nvidia Corporation (Dec. 2008).

Aljure, D.E., Lehmkuhl, O., Rodríguez, I., Oliva, A. Flow and turbulent structures around simplified car models (2014) Computers and Fluids, 96, pp. 122-135.

F. Nicoud, F. Ducros. 1999. Subgrid-Scale Stress Modelling Based on the Square of the Velocity Gradient Tensor. Flow, Turbulence and Combustion. Volume 62, Issue 3 , pp 183-200