6th European Conference on Computational Mechanics (ECCM 6)
7th European Conference on Computational Fluid Dynamics (ECFD 7)
11-15 June 2018, Glasgow, UK

STRATEGIES FOR THE HETEROGENEOUS EXECUTION OF LARGE-SCALE SIMULATIONS ON HYBRID SUPERCOMPUTERS

Xavier Álvarez[1], **Andrey Gorobets**[2] **and F. Xavier Trias**[1]

[1] Heat and Mass Transfer Technological Center, Technical University of Catalonia
C/ Colom 11, Terrassa (Barcelona), 08222, Spain

[2] Keldysh Institute of Applied Mathematics RAS
Miusskaya Sq. 4, Moscow, 125047, Russia

Key words: Parallel CFD, SpMV, Portability, MPI + OpenMP + OpenCL, Hybrid CPU + GPU, Heterogeneous computing

Abstract. Massively-parallel devices of various architectures are being adopted by the newest supercomputers to overcome the actual power constraint in the context of the exascale challenge. This progress leads to an increasing hybridisation of HPC systems and makes the design of computing applications a rather complex problem. Therefore, the software efficiency and portability are of crucial importance. In this context of accelerated innovation, we developed the HPC^2 (Heterogeneous Portable Code for HPC). It is a portable, algebra-based framework for heterogeneous computing with many potential applications in the fields of computational physics and mathematics, such as modelling of incompressible turbulent flows. In its application to CFD, the algorithm of the time-integration phase relies on a reduced set of only three algebraic operations: the sparse matrix-vector product, the linear combination of vectors and the dot product. This algebraic approach combined with a multilevel MPI+OpenMP+OpenCL parallelisation naturally provides modularity and portability. In this work, we present the strategies for the efficient heterogeneous execution of large-scale simulations on hybrid supercomputers that are part of the HPC^2 core. The performance is studied in detail for the major computing kernel, the sparse matrix-vector product, using a sparse matrix derived from a simulation on a hybrid unstructured mesh and up to 32 nodes of a hybrid CPU+GPU supercomputer.

1 INTRODUCTION

Continuous enhancement in hardware technologies enables scientific computing to advance incessantly to reach further aims. After hitting petascale speeds in 2008, several organisations and institutions began the well-known global race for exascale high-performance computing (HPC). Thenceforth, hardware developers have been facing two significant challenges. Firstly, the energy efficiency of the exascale systems ought to be augmented by two orders of magnitude respect to the earliest petascale machines. Secondly, the memory bandwidth must be increased to satisfy the demands of the scientific computing community. The common FLOP-oriented architectures (*i.e.* very high, and growing FLOPS to memory bandwidth ratios) are not efficiently dealing with most of the algorithms used in scientific computing; they barely reach 3% of their peak performance as shown in the HPCG Benchmark [1].

In consequence, massively-parallel devices of various architectures are being incorporated into the newest supercomputers. This progress leads to an increasing hybridisation of HPC systems and makes the design of computing applications a rather complex problem. The computing operations that form the algorithms, the so-called kernels, must be compatible with distributed- and shared-memory SIMD and MIMD parallelism and, more importantly, with stream processing (SP), which is a more restrictive parallel paradigm. Initially, the GPU-only implementations proved to be more energy efficient than the CPU-only did [2], even though they let the majority of CPU cores on the hybrid nodes to remain idle. Heterogeneous implementations rapidly became popular since they can target a wide range of architectures and combine different kinds of parallelism, engaging all the computing hardware available on the node. For instance, the MPI+OpenMP+CUDA implementation in [3] provides high scalability on up to 1024 hybrid nodes. However, only depending on the proprietary NVIDIA CUDA framework for handling GPUs leads to a loss of portability while, considering the enormous complexity of porting existing codes, the software efficiency and portability is of crucial importance. Therefore, fully-portable implementations such as the MPI+OpenMP+OpenCL in [4] are preferred in this work.

In this context of accelerated innovation, making an effort to design modular applications composed of a reduced number of independent and well-defined code blocks is worth it. On the one hand, this helps to reduce the generation of errors and facilitates debugging. On the other hand, modular applications are user-friendly and more comfortable for porting to new architectures (the fewer the kernels of an application, the easier it is to provide portability). Furthermore, if the majority of kernels represent linear algebra operations, then standard optimised libraries (*e.g.* ATLAS, clBLAST) or specific in-house implementations can be used and easily switched.

Nevertheless, the design of modular frameworks requires a long-term, global strategy to ensure that the pieces fit correctly. If you take a look at the studies in numerical methods that mimic the properties of the underlying physical and mathematical models, most of them use an operator-based formulation because of its power to analyse and construct accurate discretisations [5, 6]. Such a formulation encouraged Oyarzun et al. [7] to implement an algebra-based CFD algorithm for simulation of incompressible turbulent flows. Roughly, the approach consists in replacing traditional stencil data structures and sweeps by algebraic data structures and kernels. As a result, the algorithm of the time-integration phase relies on a reduced set of only three basic algebraic operations: the sparse matrix-vector product, the linear combination of vectors and the dot product. Consequently, this approach combined with a multilevel MPI+OpenMP+OpenCL parallelisation naturally provides modularity and portability. Furthermore, in our previous work, we generalised the concept of the framework to extend its applications beyond CFD;

we presented in [8] the HPC^2 (Heterogeneous Portable Code for HPC), a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics.

The most time-consuming operation in our framework is the sparse matrix-vector product (SpMV), which represents up to 80% of the computational time of simulation as shown in [7]. It widely receives much attention because it is prevalent and even essential in many computing applications. Significant effort has been made in many works to adapt sparse matrix storage formats for different architectures and matrix properties. For instance, see [9, 10, 11, 12]. This key operation is a bottleneck in scientific computing because it is a memory-bounded operation with a very low arithmetic intensity and it leads to indirect memory accesses with unavoidable cache misses. Additionally, it is very challenging in parallel computing because it may involve both inter- and intra-node data exchanges as a result of the domain decomposition approach. Thus, hiding this expensive communication overhead behind the computations is critical. The benefits of the overlap of communications and computations on GPUs for the SpMV are demonstrated in [13]. The heterogeneous execution of the SpMV on hybrid CPU+GPU systems is studied in [14], showing a notable gain; notwithstanding, that study is restricted to a single node with a single NVIDIA GPU.

In this work, we present the strategies for the efficient heterogeneous execution of large-scale simulations on hybrid supercomputers that are part of the HPC^2 core. Firstly, the multilevel domain decomposition is proposed as the optimal method for distributing the workload across the HPC system. Secondly, both the multithreaded simple and double overlap execution diagrams are described. Finally, the heterogeneous performance is studied in detail for the major computing kernel, the SpMV, using a sparse matrix derived from a simulation on a hybrid unstructured mesh and up to 32 nodes of a hybrid CPU+GPU supercomputer.

2 STRATEGIES FOR EFFICIENT HETEROGENEOUS COMPUTING

The HPC^2 is a fully-portable, algebra-based framework with many potential applications in the fields of computational physics and mathematics. Details about its application to CFD can be found in [8]. The simulations are to be executed on a hybrid HPC system that consists of multiple computing nodes interconnected via a communication infrastructure. Hence, the optimal distribution of the workload across the system is of great importance for attaining maximum performance. In our framework, a single MPI process is assigned to each hybrid node. The workload distribution is fulfilled with a multilevel domain decomposition. Hence, the MPI processes must handle the computing hardware through OpenMP parallel regions and multiple OpenCL queues. Both the multithreaded simple and double overlap execution diagrams are described to that end.

2.1 The role of the SpMV in large-scale simulations

Large sparse matrices often appear when numerically solving partial differential equations. Hence, the sparse matrix-vector product (SpMV) is a widespread operation in the scientific computing community. The sparse pattern of a matrix (*i.e.* the distribution of the non-zero coefficients) typically arises from the spatial discretisation of a computational domain. The discretised domain is a finite set of objects in which some pairs are in some sense related (such as mesh nodes, cells, faces, vertices, etc.). The couplings of an element (represented by the non-zero coefficients within its row) depend on the numerical method utilised.

In our algebra-based approach, the SpMV kernel represents the 80% of the computational cost. In addition, this key operation is the only one among the three of the HPC^2 that requires data exchanges as a result of the domain decomposition approach. That is, the other two kernels are independent of the workload distribution. Therefore, we will only focus on the heterogeneous implementation of the SpMV from now on, considering that the one of the *axpy* and *dot* is straightforward.

2.2 First-level decomposition for distributed-memory parallelisation

By way of example, let us consider the generic discretised computational domain in Figure 1 (left). The first-level domain decomposition distributes the workload among the computing nodes (*i.e.* the MPI processes). In doing so, the elements of the discretised domain are assigned to subdomains using a partitioning library (*e.g.* ParMETIS [15]) that fulfils the requested load balancing and minimises the number of couplings between cells of different subdomains. As a result, the first-level decomposition classifies subdomain elements into *inner* and *interface* categories as shown in Figure 1 (right). Namely, *interface* elements are those coupled with elements from other subdomains. Consequently, those other's adjacent elements form a *halo*. A communication between parallel processes is required to update the *halo* before a kernel processes the *interface*.

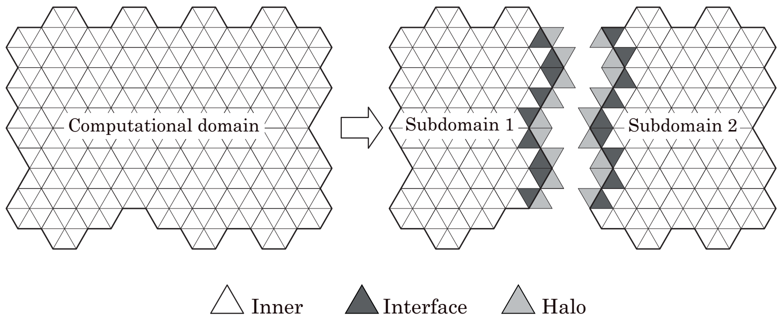

Figure 1: Representation of a generic discretised domain (left). First-level decomposition of the domain among two MPI processes (right).

2.3 Second-level decomposition for intra-node parallelisation

First-level subdomains are decomposed further to distribute the workload of each MPI process among its computing hardware, such as multiple CPUs (called *host*) and coprocessors of different kinds (called *devices*). This second-level decomposition must conform to the actual performance of the hardware for the sake of load balancing. We propose in Figure 2 two different decomposition strategies of the same first-level subdomains. On the left side, we aim at minimising the number of couplings. On the right, we isolate the *devices* from the external communications. In this case, the *host* partition takes the whole first-level *interface* plus a set of *inner* nodes. As a result, at the second level, the *interface* and *halo* elements are classified as (1) external ones coupled with other subdomains of the first-level decomposition (grey-coloured), and (2) internal ones that only participate in the intra-node exchanges (yellow-coloured).

The multilevel decomposition reduces the volume of the external communications several times since only the *interface* of the first-level decomposition (grey-coloured) remains external. Put another way; if the workload were initially distributed with flat domain decomposition by assigning one MPI process to each *device*, the partitions would not be aware of which *devices* are located on the same nodes. Therefore, the whole *interface* of each subdomain would be involved in external communications. We do not recommend it because the external *interface* assigned to a *device* with a separate memory space requires a more expensive multistage *device* → *host* → MPI → *host* → *device* communication.

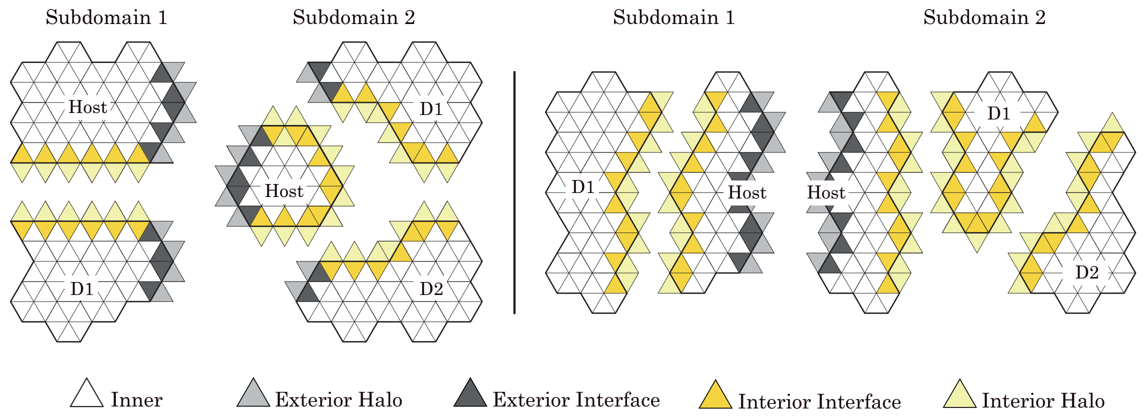

Figure 2: Two different strategies for the second-level decomposition. Minimising the number of couplings (left), isolating the *devices* from the external communications (right).

2.4 Multithreaded overlap strategies

The heterogeneous execution of the SpMV leads to many different data-management and computing operations because the complex multistage *halo* update must be concluded before computing the *interface* elements. This multistage update is an expensive operation that can critically affect the performance and scalability. Therefore, efficient strategies are required for minimising the overhead of the communications. We describe hereafter both the multithreaded simple and double overlap execution diagrams. Essentially, this execution aims at simultaneously handling operations and communications of both *host* and *device* through OpenMP parallel regions and multiple OpenCL queues.

For a better understanding of the overlap diagrams in Figure 3, and also for timing purposes, we compressed all the operations involved in the SpMV into five distinct blocks (outlined in Table 1). The *host* blocks (white-coloured) are blocking tasks in the sense that the assigned OpenMP threads must finish them before proceeding. In contrast, the *device* blocks (grey-coloured) are non-blocking; that is, the assigned OpenMP thread only submits the task to an OpenCL queue, then continues with the following. In D2H and H2D blocks, the use of mapped, pinned-memory OpenCL intermediate buffers is necessary; such pinned buffers are needed for DMA transfers, which can be overlapped with computations.

Table 1: Description of the operational blocks composing the execution diagrams in Figure 3.

Block	**Description**	**Timer**
INN	The SpMV kernel is launched only for the *inner* elements. The *inner* domain can be overlapped with the *halo* update because it is not coupled with the *halo*.	t_{inn}
D2H	*Device-to-host* download. Both the *device*'s internal and external *interface* data is packed into intermediate buffers, then copied to *host*.	t_{d2h}
MPI	Inter-node exchanges. The send messages are posted with point-to-point, non-blocking MPI_Isend calls, and receive messages with MPI_Irecv for the incoming data. Finally, MPI_Waitall is used for synchronisation.	t_{mpi}
H2D	*Host-to-device* upload. Both the *device*'s internal and external *halo* data is packed into intermediate buffers, then copied to the *device*.	t_{h2d}
IFC	The SpMV kernel is launched only for the *interface* elements as soon as the *halo* update is completed. In the end, a synchronisation barrier is included for *host* and *devices*.	t_{ifc}

The simple overlap diagram is shown in Figure 3 (left). In this mode, the *halo* update is overlapped with the INN and IFC blocks. Two threads are created in the outer OpenMP region: one for *host* computations and another for managing the *device*'s OpenCL queues. The *device*'s outer thread spawns an OpenMP nested region with as many threads as *devices* (th_d). The *host* thread executes the *host* computations within an OpenMP nested region engaging those threads still available (th_h). In this simple overlap method, the *devices* are involved in external communications. Hence, the H2D block must start right after the MPI synchronisation. Assuming that all *devices* perform equal, the overall computational time of the simple overlap can be estimated as

$$t_{sov} = max(t^h_{inn}, (t^d_{inn} + t_{d2h} + t_{mpi})) + max(t^h_{ifc}, (t_{h2d} + t^d_{ifc})). \tag{1}$$

Thus, since the time depends on the maximum values in between different blocks, a proper load balancing becomes very important.

The double overlap diagram is shown in Figure 3 (right). The main difference is that, in this case, the external MPI communications are performed simultaneously with the internal D2H and H2D exchanges. To do so, the second-level decomposition must have isolated the *devices* from the external *interface* as shown in Figure 2 (right). This way, the MPI block becomes independent of the D2H, and consequently, H2D is independent of MPI. Nevertheless, *host* IFC is bigger in this case. Hence, the double overlap is beneficial only if the *host* computations are faster than the internal communications. The overall computational time of the double overlap can be estimated as

$$t_{dov} = max(t^{h}_{inn}, max(t_{mpi}, (t^{d}_{inn} + t_{d2h} + t_{h2d}))) + max(t^{h}_{ifc}, t^{d}_{ifc}). \tag{2}$$

Besides, the synchronous execution scheme may be relevant for test and comparisons. It is not shown in Figure 3 due to its simplicity. Essentially, the idea is to complete the *halo* update first, then proceed with computations. The overall computational time of the synchronous mode is straightforward:

$$t_{syc} = t_{d2h} + t_{mpi} + t_{d2h} + t_{inn} + t_{ifc}. \tag{3}$$

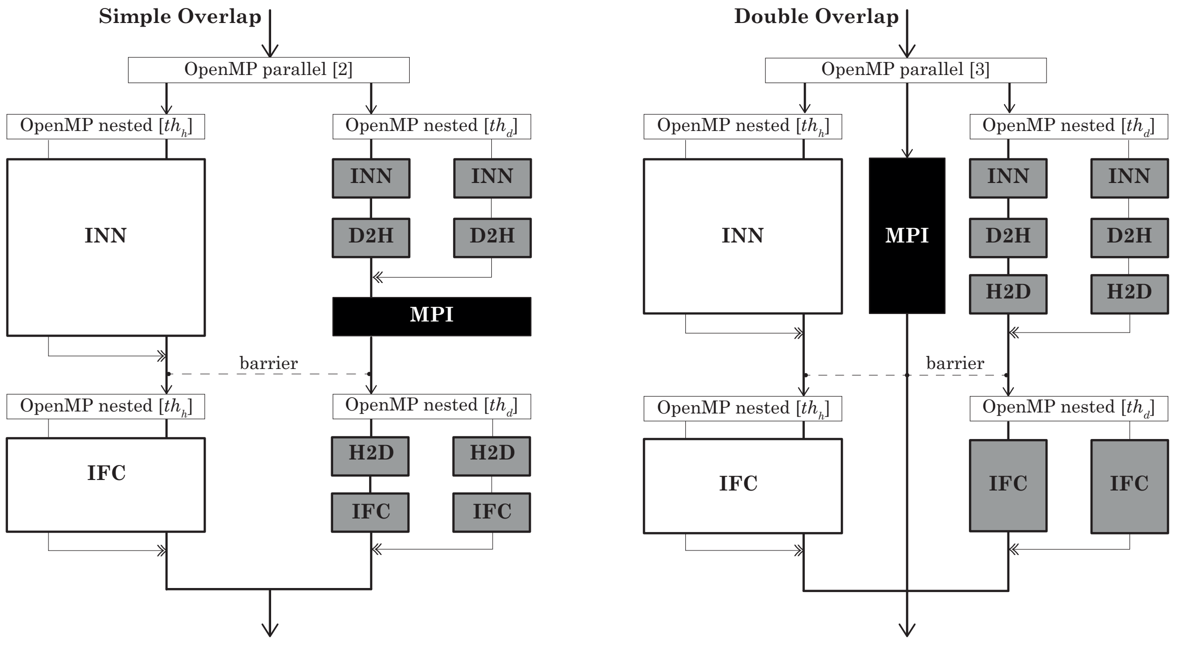

Figure 3: Multithreaded execution diagrams for simple overlap (left), double overlap (right). The white-coloured blocks correspond to *host*; the grey-coloured ones correspond to *devices*.

3 HETEROGENEOUS PERFORMANCE STUDY

The sparse matrix used in this study is derived from a symmetry-preserving discretisation [5] on an unstructured hex-dominant mesh with approximately 10M cells. Therefore, the majority of rows hold seven non-zero coefficients. The adapted block-transposed version of ELLPACK sparse storage format [9] is used for the study. This format provides a uniform aligned memory access with coalescing of memory transactions.

The benefits of the heterogeneous CPU+GPU execution of the SpMV are measured on the Lomonosov-2 hybrid supercomputer. Its nodes are equipped with a 14-core Intel E5-2697v3 CPU and an NVIDIA Tesla K40M GPU. The performance comparison for the CPU-only, GPU-only and heterogeneous executions on a single node is shown in Figure 4. The heterogeneous mode shows a gain of 32% compared to the GPU-only mode, which corresponds to a 98% of heterogeneous efficiency compared to the sum of the performance of the CPU-only and the GPU-only modes.

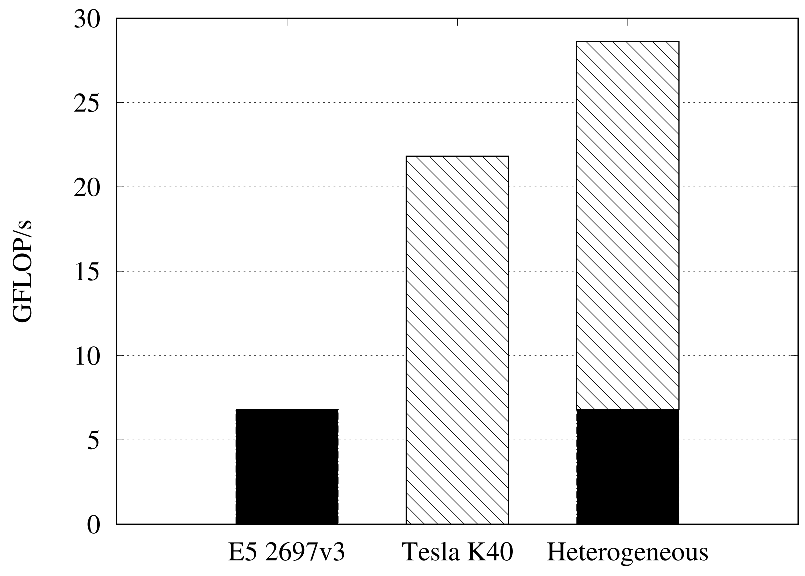

Figure 4: Single-node performance comparison for CPU-only, GPU-only and heterogeneous modes.

The strong scalability results in Figure 5 show that the simple overlap strategy presented in this work notably improves the performance by hiding the communications. However, the scalability decays faster in the heterogeneous mode compared to the GPU-only mode. This leak occurs because in the former, the computational load per GPU is smaller and the communication load is higher than of the latter. Besides, in the heterogeneous mode, the CPU is loaded with computations which may interfere with MPI library routines. Therefore, the overlapping operational range gets reduced.

Finally, the sustained performance for the different execution modes is shown in Figure 6. It can be seen that despite a little weaker scalability, the heterogeneous mode outperforms the GPU-only mode. However, this advantage decays again with the number of nodes because the CPU increasingly gets more involved in communications and the load per GPU decreases.

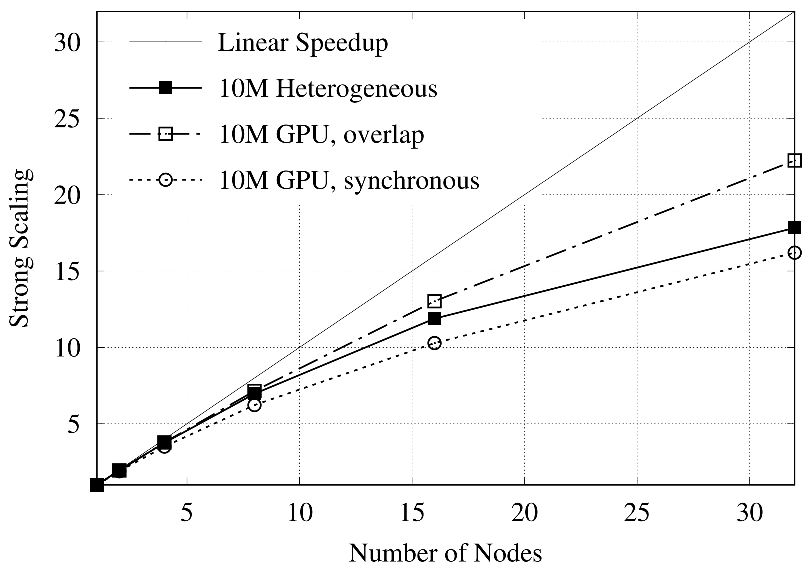

Figure 5: Strong scalability study of the SpMV for different execution modes.

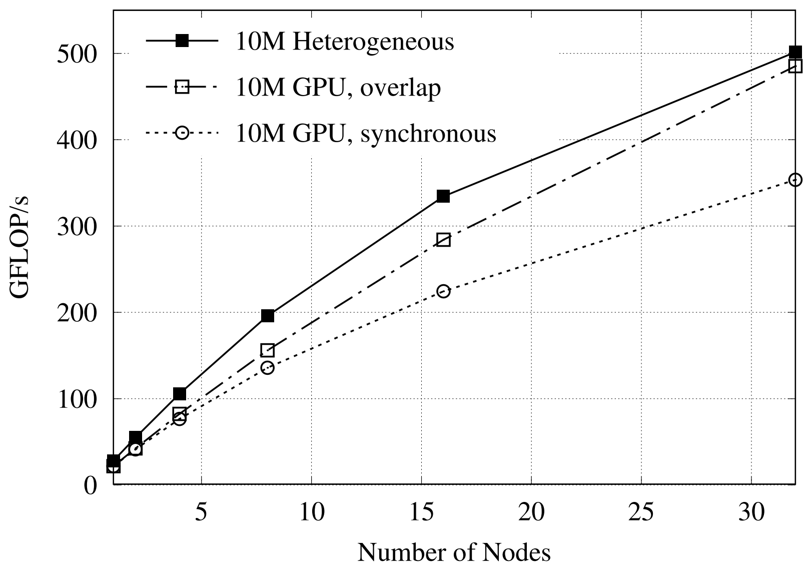

Figure 6: Performance comparison of the SpMV for different execution modes.

4 CONCLUSIONS

An algebra-based framework with a heterogeneous MPI+OpenMP+OpenCL implementation has been presented. This approach naturally provides modularity and portability; it can target a wide range of architectures and combine different kinds of parallelism, engaging all the computing hardware available on the node. Considering the increasing hybridisation of HPC systems, this appears to be very relevant.

The strong scalability study shows that the benefit of the heterogeneous execution of the SpMV decreases with the number of nodes. Therefore, to efficiently run heterogeneous large-scale simulations, the load per device must be enough to guarantee the adequate performance of the devices and the sufficient contribution of the host. Heterogeneous performance studies must be carried out to determine the optimal workload distribution. To that end, our algebra-based framework appears to suit very well since a single kernel (the SpMV) is a key representative of the overall performance.

Acknowledgments

The work has been financially supported by the *Ministerio de Economía y Competitividad*, Spain (ENE2014-60577-R). X. Á. is supported by a *FI* predoctoral contract (FI_B-2017-00614). F. X. T. is supported by a *Ramón y Cajal* postdoctoral contract (RYC-2012-11996). This work has been carried out using computing resources of the the Center for collective use of HPC computing resources at Lomonosov Moscow State University; the authors thankfully acknowledge these institutions. The authors also wish to thank Ms. E. Pino for her helpful illustrations.

REFERENCES

[1] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," Tech. Rep. SAND2013-4744, Sandia National Laboratories, 2013.

[2] P. Zaspel and M. Griebel, "Solving incompressible two-phase flows on multi-GPU clusters," *Computers & Fluids*, vol. 80, pp. 356–364, jul 2013.

[3] C. Xu, X. Deng, L. Zhang, J. Fang, G. Wang, Y. Jiang, W. Cao, Y. Che, Y. Wang, Z. Wang, W. Liu, and X. Cheng, "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer," *Journal of Computational Physics*, vol. 278, pp. 275–297, dec 2014.

[4] A. Gorobets, S. Soukov, and P. Bogdanov, "Multilevel parallelization for simulating compressible turbulent flows on most kinds of hybrid supercomputers," *Computers & Fluids*, mar 2018 (published online).

[5] F. Trias, O. Lehmkuhl, A. Oliva, C. Pérez-Segarra, and R. Verstappen, "Symmetry-preserving discretization of Navier–Stokes equations on collocated unstructured grids," *Journal of Computational Physics*, vol. 258, pp. 246–267, feb 2014.

[6] K. Lipnikov, G. Manzini, and M. Shashkov, "Mimetic finite difference method," *Journal of Computational Physics*, vol. 257, pp. 1163–1227, jan 2014.

[7] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, "Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers," *International Journal of Computational Fluid Dynamics*, vol. 31, pp. 396–411, oct 2017.

[8] X. Álvarez, A. Gorobets, F. Trias, R. Borrell, and G. Oyarzun, “HPC^2 – A fully-portable, algebra-based framework for heterogeneous computing. Application to CFD,” *Computers & Fluids*, feb 2018 (published online).

[9] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures,” in *High Performance Embedded Architectures and Compilers* (Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, eds.), pp. 111–125, Springer Berlin Heidelberg, 2010.

[10] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, “Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications,” in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 781–792, nov 2014.

[11] J. L. Greathouse and M. Daga, “Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format,” in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, nov 2014.

[12] W. Liu and B. Vinter, “CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 339–350, ACM Press, mar 2015.

[13] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, “MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner,” *Computers & Fluids*, vol. 92, pp. 244–252, mar 2014.

[14] W. Yang, K. Li, and K. Li, “A hybrid computing method of SpMV on CPU–GPU heterogeneous computing systems,” *Journal of Parallel and Distributed Computing*, vol. 104, pp. 49–60, jun 2017.

[15] D. Lasalle and G. Karypis, “Multi-threaded Graph Partitioning,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 225–236, may 2013.