14th World Congress in Computational Mechanics (WCCM)
ECCOMAS Congress 2020
Virtual Congress: 11 – 15 January 2021
F. Chinesta, R. Abgrall, O. Allix and M. Kaliske (Eds)

ADAPTING COMPLEX AND CLUMSY CFD CODE TO RAPIDLY CHANGING SUPERCOMPUTING REALITIES

ANDREY GOROBETS[1]

[1] Keldysh Institute of Applied Mathematics (Russian Academy of Sciences),
4, Miusskaya Sq., Moscow, 125047, Russia
andrey.gorobets@gmail.com, http://caa.imamod.ru/

Key words: Supercomputer simulation, heterogeneous computing, compressible flow, unstructured mesh, MPI, OpenMP, OpenCL

Abstract. This work is devoted to acceleration and upgrade of the CFD code NOISEtte for scale-resolving simulations of compressible turbulent flows using edge-based high-accuracy methods on unstructured hybrid meshes. Attempts to extend the baseline multilevel MPI+OpenMP parallelization towards GPU-based hybrid systems have faced the problem: the code is too complex. It is an in-house research code with plenty of numerical methods, schemes, models, most of which are experimental and are not used in practical simulations. This chaotic zoo leads to excessive conditional branches, switches, redundant functional calls that slow down computations. Although the parallel algorithm is fully adapted to the stream processing paradigm, such an immense amount of code is too difficult to port efficiently to OpenCL or CUDA and maintain it in consistency with the CPU version. An approach to survive in the process of adaptation to hybrid systems has been elaborated. It consists of various components, such as creation of a simplified configurations, combining different stages of the algorithm in order to reduce memory traffic, collapsing multiple functions in one function without branches and switches, mixing single and double precision, etc. As a result, the upgraded code is about twice as fast on CPUs and can use GPUs from different manufacturers - AMD, NVIDIA, Intel through the OpenCL standard.

1 INTRODUCTION

CPUs are currently trying to incorporate faster memory channels (such as Open Memory Interface of IBM), and manycores with onboard high-bandwidth memory (such as Intel Xeon Phi formerly or Fujitsu A64FX currently) are competing with GPUs. Meanwhile, high performance computing (HPC) programmers for scientific applications have to make a difficult choice – which potential architectures and software frameworks to rely on. What if GPUs win and CPU performance will be incomparably smaller? What if CPUs strike back and reach parity with GPUs? It seems better not to bet on either side and create codes that can exploit any winner effectively. Currently CPUs are potentially much slower than high-end GPUs, and there are many GPU-based systems available. Therefore, in order to use those resources, simulation codes for CPUs must be somehow extended to GPUs. Once a version for GPU is available, both kinds of devices, CPUs and GPUs, can be engaged (even simultaneously), hence, no matter which of the kinds wins. In this context, the present work is

devoted to acceleration and upgrade of the computational fluid dynamics (CFD) code NOISEtte towards heterogeneous computing on CPUs and GPUs. This code is designed for scale-resolving simulations of compressible turbulent flows using edge-based high-accuracy methods on unstructured mixed-element meshes [1,2].

GPU computing is rather widely used in applications, although certainly not to the same extent as computing on CPUs. Many examples of simulation codes capable of using multiple GPUs can be found in the literature. Majority of simulation codes use CUDA, a proprietary framework only for NVIDIA GPUs. For instance, simulating compressible flows on multiple GPUs using an implicit method with a preconditioned BiCGSTAB solver is shown in [3], which is very close to what is being done in the present work. An example of large multi-GPU simulation of aerodynamic problems can be found in [4]. In that work, rather compute-intensive high-order numerical schemes are used, which allows obtaining very high sustained performance. In contrast, the schemes used in the present work are much cheaper computationally, so the algorithm is clearly memory-bound. Another example of a CUDA-based supercomputing code for aerodynamics problems can be found in [5]. Co-execution of scale-resolving simulations on CPUs and GPUs is demonstrated there. However, it is an "incompressible" code. The present work also considers a simulation code on unstructured meshes for aerodynamics problems. But our code is “compressible”, therefore, it can be applied not only to generate Q-criterion plots of a flow around an aircraft in landing conditions, but mainly to obtain reasonably accurate results in a wide range of Mach numbers. Another important difference is that the OpenCL computing standard is used instead of CUDA in the present work in order to reach maximal portability. This standard is supported by NVIDIA as well as by other major GPU vendors, such as AMD and Intel.

The proposed MPI+OpenMP+OpenCL parallelization can exploit most kinds of existing supercomputer architectures. Examples of similar OpenCL-based approaches with static load balancing can be found, for instance, in [6] for compressible flows and in [7] for incompressible flows, respectively. Heterogeneous execution of simulations of incompressible flows with dynamic load balancing between CPUs and GPUs is presented in [8]. However, in these heterogeneous implementations [6–8], one MPI process can engage CPU cores and multiple accelerators of a hybrid node by distributing roles among OpenMP threads: some threads compute on CPUs, some threads execute OpenCL kernels on devices and perform data transfer. In the present work, a simpler approach is chosen: one MPI process per device, either a CPU or an accelerator, so the roles are distributed at the MPI level.

The baseline version of the NOISEtte code had multilevel MPI+OpenMP parallelization [9] for tens of thousands of CPU cores that works fine on manycores such as Intel Xeon Phi. However, attempts to improve its performance and extend it to GPU-based hybrid systems have faced the problem: the code is too complex. It is an in-house research code with plenty of numerical methods, schemes, models, most of which are experimental and are not used in practical simulations. This chaotic zoo, combining 2D and 3D cell- and vertex-centered approaches, leads to numerous conditional branches, switches, redundant functional calls that slow down computations. Although the baseline parallel algorithm is fully adapted to the stream processing paradigm, this immense amount of code is too difficult to port efficiently and maintain. An approach to survive in the process of adaptation to hybrid systems has been elaborated. It consists of various components described in the following sections, such as improvement of reliability, creation of simplified configurations of the code for resource-

intensive simulations, combining different stages of the algorithm in order to reduce memory traffic, mixing single and double precision, creation of internal debug and testing infrastructure for development and maintenance of OpenCL kernels.

2 PROBLEM STATEMENT

The code under consideration is the NOISETTE CFD code for solving aerodynamics and aeroacoustics problems using scale-resolving methods in both research and applications. It is based on solving Navier–Stokes equations for compressible viscous flows using high-accuracy numerical schemes on unstructured mixed-element meshes [1, 2]. Hybrid RANS-LES approaches, such as DES and its variants [10], are used for turbulence modeling. Explicit Runge-Kutta and implicit Newton-based schemes are used for time integration.

The code is written in C++. The baseline parallel implementation for CPUs has combined MPI+OpenMP parallelization based on multilevel decomposition. It is well suited for big supercomputers made of multi-socket nodes with manycore CPUs. Detailed description of the parallelization for CPUs and its parallel efficiency in simulations using more than 10 thousand cores is demonstrated in [9]. The OpenMP parallelization can deal efficiently with hundreds of threads, targeting such devices as Intel Xeon Phi. It was giving about ×2 speedup on a 68-core Intel Xeon Phi 7250 compared to a 16-core CPU Intel Xeon v4.

The algorithm is rather simple to parallelize, it fits MIMD parallelism and stream processing parallel paradigm (with certain minor modifications). Thus, it seemed easy to implement it on GPUs. But the problem appeared to be not the parallel algorithm but the code itself. It appeared difficult to make it work fine with a reasonable effort. Making it reliable, modifiable and easy to maintain is not an easy task. The problem is its complexity and clumsiness, which comes from combining two contradictory things: 1) a playground for development of new numerical methods, which implies having many numerical methods inside, together with an infrastructure for implementing, testing and comparing them; 2) a tool for industrial applications, which needs high computing performance, efficiency and reliability.

A heterogeneous implementation for GPUs complicates the things significantly, so intrinsic simplicity of the code is required. The proposed approach mainly consists in fighting with complexity and achieving reliability, which appeared to be difficult to preserve in an intensively modified heterogeneous code.

3 IMPROVEMENT OF RELIABILITY

Reliability from a software perspective assumes that errors can hardly penetrate into production, and if they do, those nasty bugs are easy to localize and catch. Of course, there must be sufficient quality assurance (QA) coverage. Apart from QA, correctness checks can be massively placed throughout the code to prevent bugs. But those assertions may introduce significant overhead in terms of performance, slowing down resource-intensive simulations. Therefore, before starting work on the GPU implementation, a "**safe-mode**" configuration was introduced, which seems a rather common thing. It allows extensive correctness checks to be disabled in the release configuration and easily enabled in the case if any problem appears.

Firstly, functions of the code were split into two categories: low-level functions, which process a single mesh object (cell, node, face, etc.), and high-level functions that process sets

of many mesh objects. Those low-level functions form the performance-critical part of the code. For such high-frequency calls, correctness checks produce notable overhead. In contrast, in high-level functions, the performance impact of checks is negligible due to the high computational load. In accordance with this, the checks were divided into two parts for high and low levels, respectively. The checks in low-level functions can be disabled in the release configuration. This is implemented with two macro wrappers, such as ASSERT(<condition>, <diagnostic message>) for the high level, which is always enabled, and similar SAFE_ASSERT, which is only enabled in the safe-mode configuration. Most arrays are also accessed with explicit correctness checks in the safe-mode configuration. For instance, the basic array containers in the code have access operators (operator[] in C++) with the range check in the safe mode. In the case of a wrong access, those checks allow to identify the place and particular array explicitly without using any debug tools, which work orders of magnitude slower and are hardly applicable if an error appears in a big simulation. Further details can be found in [11].

4 SIMPLIFICATION AND IMPROVEMENT OF PERFORMANCE

Then, to start with GPUs, we need to find a solution for the problem of the code's superfluous complexity. Apart from difficulties in dealing with the heterogeneous implementation, this complexity also leads to a lot of branches, switch operators, inner function calls at low level, which, in turn, produce significant slowdown in terms of performance. Fighting this complexity, on the one hand, is needed to approach GPU computing, and, on the other hand, it appears to significantly speed up the code on CPUs.

In main low-level functions, such as the computation of fluxes through a face between control volumes, there are a lot of switch operators and nested function calls for different versions of methods and implementations. Figure 1 shows how this might look.

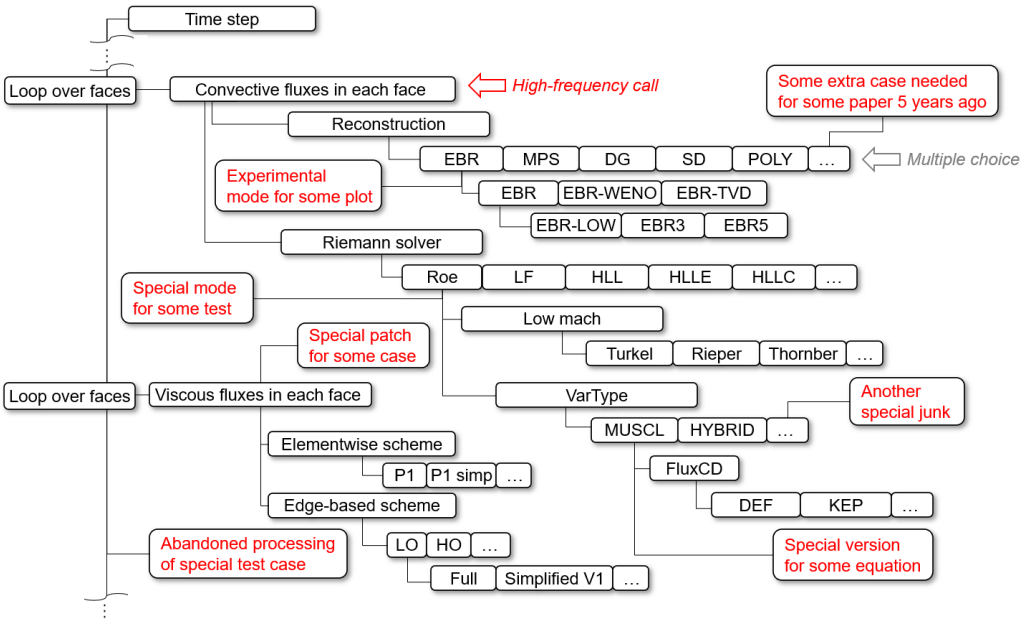

Figure 1: Outline of the call tree structure that indicates superfluous complexity, which affects performance

The problem of such a superfluous complexity could be easily solved just by throwing away all this unnecessary and obsolete functionality, but not in the case of a scientific code. In such a code, it becomes a social problem: scientists take it as a personal offence if one tries to remove their abandoned stuff from the code. But this social problem has a rather simple software solution. In order to hide this overhead on superfluous branching at the low level, the "**playground**" configuration was introduced. All the numerical schemes, models, reconstruction options, Riemann solvers, etc., used in some research and comparative studies but not in practical resource-intensive applications, were hidden under a macro definition. This definition is only enabled in the playground configuration. Doing so, a significant part of the code can be dropped in the release configuration, as shown in Figure 2.

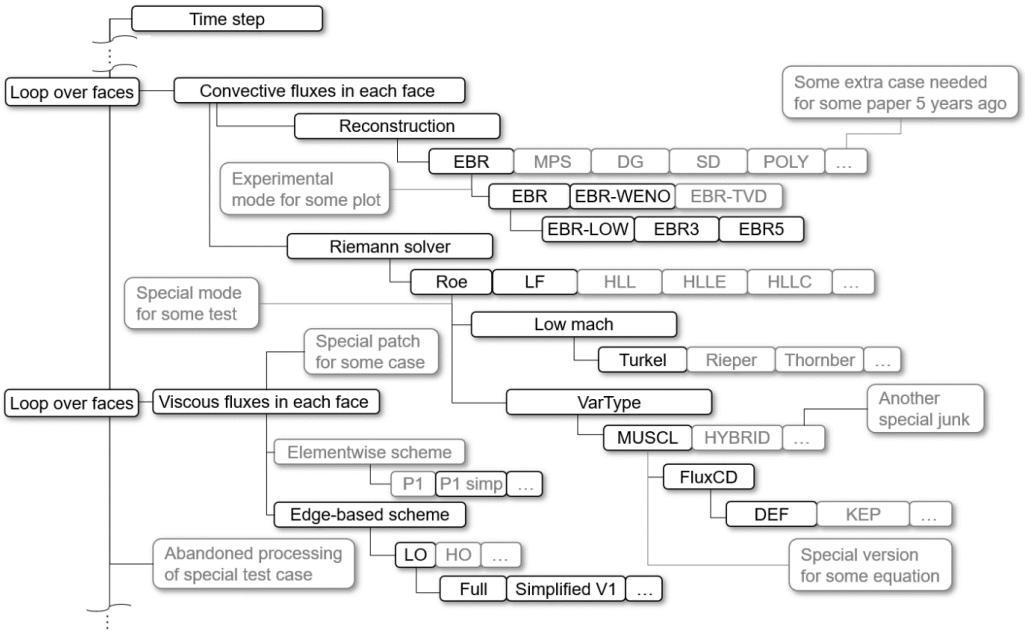

Figure 2: The playground vs release configurations: most experimental and research functionality available in the playground configuration is disabled (grayed out) in the release build

Furthermore, it appeared that if we combine for some particular numerical method settings all the inner functions, such as reconstruction of variables, Riemann solver, low-Mach preconditioner, viscous fluxes calculation, etc., into one big function without branching and nested function calls, then it works nearly twice faster (sequential execution, explicit scheme). To get these performance benefits, we created such combined functions for those configurations, which are most commonly used in resource-intensive simulations. Basically, there are two combined functions – for subsonic flows (schemes EBR3, EBR5 [1], with low-Mach preconditioner, without limiters for discontinuities) and for supersonic flows based on WENO reconstruction (EBR-WENO3, EBR-WENO5 [2]). These combined versions of the main flux calculation functions (see Figure 3) are called "**business-lunch**" in an analogy with a restaurant, where usually there is a big menu at full price and long preparation time and a lunch menu with few sets of dishes, which is served much faster and cheaper.

Apart from saving time on function calls and branching, significant improvement has been achieved by combining convective and viscous fluxes in one function in the case of an

implicit scheme. Separate functions have to access the very large Jacobi matrix twice. Combining these expensive memory accesses into one has significantly reduced memory traffic. Furthermore, since GPU memory is very limited, we developed a simplified version of the viscous fluxes calculation method by reducing the number of coefficients and using mixed single and double precision floating point formats. Details about this new method to compute viscous fluxes much cheaper can be found in [12].

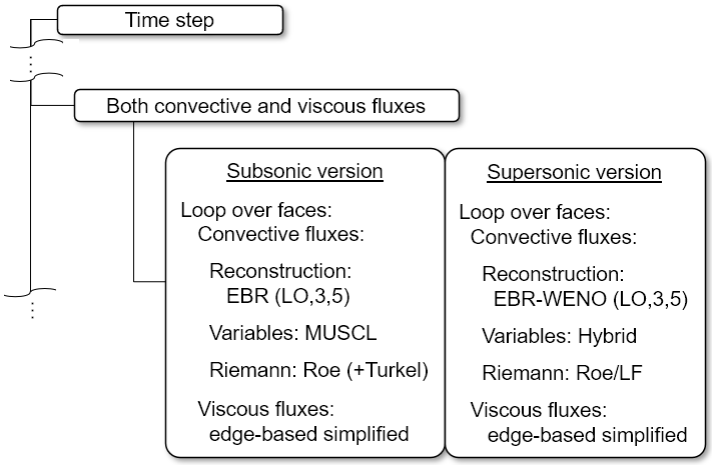

Figure 3: The "business-lunch" configuration: combining many functions inside the flux calculation stage in one in order to save time on function calls, branching and memory access

In resource-intensive applications, this "business-lunch" configuration works about 15–20% faster than the baseline full version, called "playground". Using the simplified method for viscous fluxes and mixed single-double precision (the Jacobi matrix and some memory-consuming coefficients for viscous fluxes are in single precision) gives further speedup about 1.7 times. In total, the new version works about twice faster in simulations on CPUs with MPI+OpenMP parallelization.

These improvements in reliability and significant simplification of major computing routines have allowed us to proceed with implementation for GPUs.

5 IMPLEMENTATION FOR GPU COMPUTING

The OpenCL computing standard was chosen for its portability in order to use GPUs of NVIDIA, AMD, Intel, as well as manycore CPUs. Most computing kernels were "naïvely" ported by taking loop bodies of operation over sets of mesh objects and replacing the loop counters with the work-item number. However, several performance-critical kernels were significantly modified. The changes mainly consisted in making each formerly loop iteration processed by multiple work items in order to increase occupancy of compute units. Among those kernels are the sparse matrix-vector product (SpMV) with a small-block matrix, the calculation of gradients, and some more kernels. More fundamental changes were needed for reduction operations (norms, minimum, maximum, dot product) and other operations with data interdependency between loop iterations. In particular, the calculation of fluxes through faces between cells was adapted to the stream processing paradigm by decomposing this operation into two kernels, one for calculation of fluxes and the other for summation of fluxes

into sells, using an intermediate storage for fluxes in faces.

It must be noted that the OpenCL kernel code is configured at runtime of the CPU program. Preprocessor definitions are extensively used for that purpose. This approach allows to eliminate most branches and switch operators inside kernels, which slow the code down a lot, and to use inner loops with constant range, which are more efficiently unrolled by the compiler.

Finally, to improve reliability, each OpenCL kernel is equipped with a consistency check that compares results of its CPU counterpart ensuring that there is no discrepancy. Before running any simulation on GPUs, each kernel used for this particular configuration of the numerical method is tested against its CPU version. Then, several timesteps are performed with both CPU and OpenCL versions separately, and consistency of results of these full timesteps is checked. This automatic testing at the initialization stage ensures consistency of the CPU and OpenCL versions.

The QA procedure has been modified accordingly. The set of QA tests has been replicated in order to ensure consistency of the baseline "playground" version, the "business-lunch" version and the OpenCL version. In doing so, results of all three versions are compared within QA procedure.

Multi-GPU parallelization relies on the same infrastructure for MPI communications on the CPU side of the code. MPI exchanges have been supplemented with intranode data transfers between the CPU and devices. Of course, the overlap of computations and communications is used in order to hide the communication overhead. This overlap speeds up multi-GPU execution significantly, especially in the case of an implicit Newton-based scheme, where rather fast and computationally cheap SpMV operations in the linear solver require frequent halo updates.

The scalability of the multi-GPU version (or multi-accelerator, since it's a portable implementation) is principally unchanged compared to the baseline version for CPUs. It can potentially use hundreds of devices efficiently as well.

The OpenCL version includes the set of numerical schemes EBR-LO, EBR-3, EBR-5, EBR-WENO-LO, EBR-WENO-3, EBR-WENO-5, Riemann solvers, including Roe [13] with the Turkel preconditioner [14] for low-Mach flows, our simplified method for viscous fluxes [12], the set of RANS, LES models and hybrid approaches of the DES family (including IDDES [10]), explicit Runge–Kutta and implicit BDF1, BDF2 schemes, the preconditioned BiCGSTAB solver [15].

6 HETEROGENEOUS EXECUTION ON CPU AND GPU

The use of CPUs and GPUs on hybrid cluster nodes, so-called co-execution, is implemented by means of multilevel decomposition and spawning one MPI process per device, either CPU or GPU. This approach allows to engage efficiently both kinds of devices with minimal changes in the code. The changes mainly consist in multilevel partitioning with empirically defined weights of second-level subdomains according to the actual performance ratio between devices. Then, at the beginning of the heterogeneous execution, MPI processes running on the same node are grouped together and their roles are distributed according to their local rank in the subgroup of the node. This approach turned out to be no less effective, but more convenient than our previous approach using one MPI process per node. Formerly,

multiple devices of a hybrid node, CPUs and GPUs, were engaged by one MPI process [6, 7]. The roles were distributed among OpenMP threads: some threads computed on CPU cores, some managed the GPU workload and communications. Apart from the higher complexity, another minor drawback is that a single MPI process can hardly saturate the available network bandwidth of multiple network adapters, which are usually 2 or 4 on a modern cluster node. In the present work, multiple MPI processes per hybrid node are placed, the CPU-only and GPU-only roles are distributed among MPI processes, as shown in Figure 4. Processes with the GPU-only role occupy its GPU and a few CPU cores, the rest of the cores are given to the CPU-only processes, which spawn OpenMP computing threads on their cores. On multi-socket nodes, one CPU-only MPI process is placed at each socket to avoid the NUMA factor between sockets.

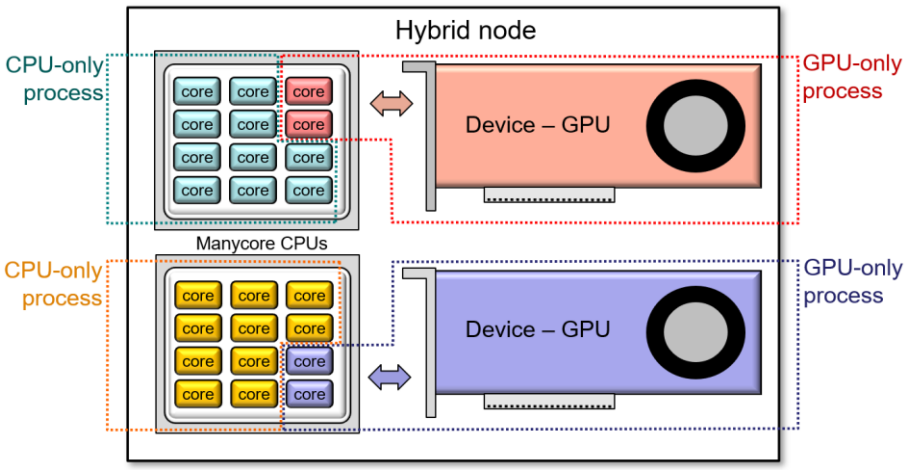

Figure 4: Heterogeneous execution on CPUs and GPUs of a hybrid node spawning one MPI process per device

It must be noted, that this co-execution is not used in practice on our local equipment (described further), because the GPUs are too powerful compared to the CPUs (about 7:1 for NVIDIA V100 vs 16-core Intel Xeon Gold), therefore, CPUs are used for parallel processing of the data exchange of devices and have no time to compute. On former equipment, the performance ratio was about 2:1 (NVIDIA K10 vs 14-core Intel Xeon v3), hence the heterogeneous mode was much more beneficial. Thus, the co-execution mode is waiting for better times when CPUs will take revenge and regain a better performance ratio with GPUs.

7 PERFORMANCE IN REAL APPLICATIONS

The only experience gained with the new version for GPUs is running urgent industry-oriented simulations on our small hybrid cluster K-60. Its hybrid nodes are equipped with two 16-core CPUs Intel Xeon Gold 6142 (120 GB/s) and four GPUs NVIDIA V100 (32 GB, 900GB/s, PCIe-4). The nodes are interconnected with the InfiniBand FDR network (2 network cards per node).

The performance has been evaluated only for the primary “business-lunch” configuration: the EBR5 scheme in space, the implicit BDF2 scheme in time, turbulence modelling IDDES [10] (with the alternative LES model [16] and the subgrid scale [17]), mixed accuracy. The

most important metric for us from a practical point of view is the actual performance ratio on GPUs vs CPUs. In other words, it is the equivalent of how many CPU cores we get from one GPU device. The memory bandwidth ratio 7.5:1 suggests the expected speedup. It appeared that the actual speedup varies around this value depending on the mesh size: the bigger is the mesh, the higher is the speedup.

For a test mesh of about 1 million nodes (flow around a sphere) the performance ratio is 7.4:1, which means that one GPU is equivalent to 119 CPU cores. All 4 GPUs of one hybrid node give us performance of 374 CPU cores. This corresponds to about 80% efficiency, which would be unachievable for such a small mesh without the overlap of computations and communications. However, notable degradation of performance indicates that the computing load is insufficient to fully hide communications. Similarly, for a twice bigger mesh (flow around a rotor blade) we get the ratio 8.7:1, so one GPU gives us 139 CPU cores. 4 devices perform as 491 CPU cores, showing parallel efficiency about 90%.

Finally, the performance is compared for a real application with a mesh of 83 million nodes. A flow around the low-pressure turbine cascade T106C [18] was considered. 3 hybrid nodes with 4 V100 GPUs each (the minimal number of devices needed to fit in memory) gave us performance of about 1500 CPU cores (nearly 125 CPU cores from 1 GPU).

Regarding the simulation, its cost is about ~2.7K CPU core hours per convective time unit. The goal of these simulations was to reproduce and study the effect of increasing the total pressure loss as the Reynolds number decreases (ranging from 500000 to 90000). These scale-resolving simulations were needed as reference solutions to tune the RANS models (with consideration of laminar-turbulent transition) for simulation of a real low-pressure turbine of a jet engine. Examples of instantaneous flow fields and some plots are shown in Figure 5.

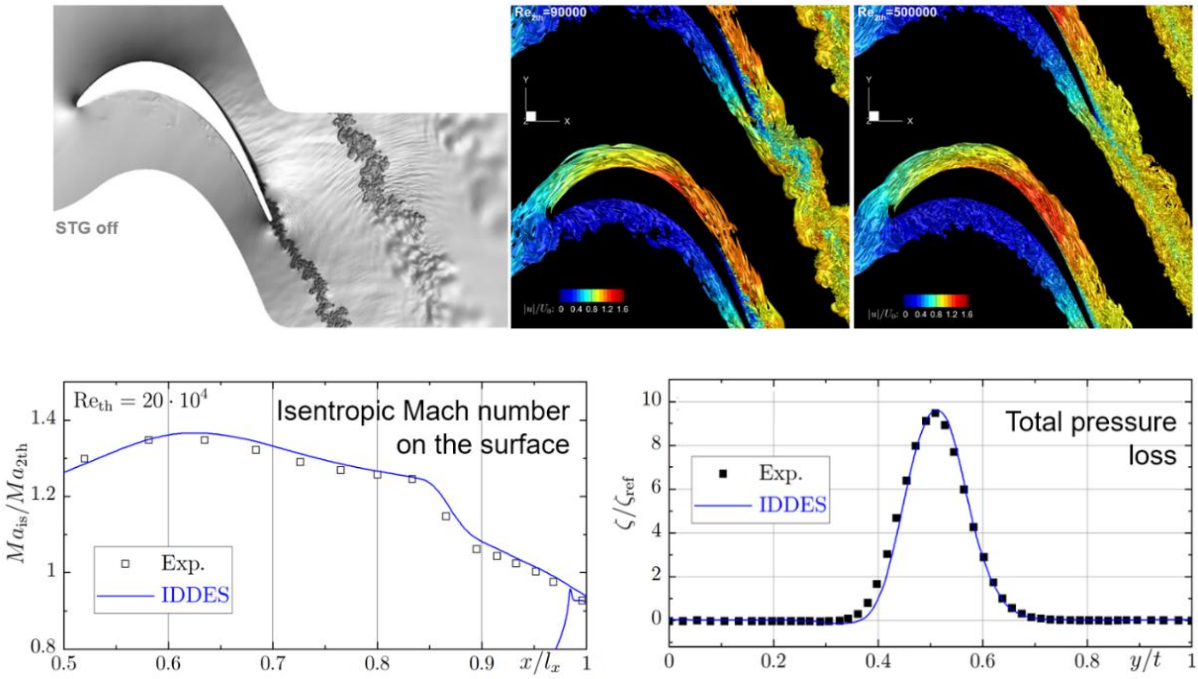

Figure 5: Examples of instantaneous flow fields (density gradient and Q-criterion, top) and some resulting plots for one of the Reynolds numbers (Re=200000, bottom)

A volume synthetic turbulence generator (based on [19]) is used upstream the blade in order to correctly reproduce the physically realistic flow conditions observed in the reference experimental study [18]. Details about these simulations will be presented in a future work by Alexey Duben et al.

Further tests with jet flows from the former numerical study [20] and the currently running scale-resolving simulations of helicopter main rotors on meshes with up to 95 million nodes using up to 20 GPUs have confirmed the performance ratio reported above.

8 CONCLUSIONS

We are happy with the obtained performance on GPUs. In multi-GPU and multi-node simulations, one GPU device NVIDIA V100 gives us the equivalent of roughly 120-140 cores of a modern processor. There is still potential for performance improvement of around 20-30%. In particular, boundary conditions need more upgrade and processing of results can be further improved. But the main problem is the limited GPU memory. We used to have at least 1 GB per core when running on CPUs. With GPUs, it is now about 4 times less considering the performance ratio. We had to significantly reduce memory consumption using mixed accuracy. And still one device with 32 GB can take only 6–7 million mesh nodes. It seems not so much for something that outperforms a hundred CPU cores.

In summary, the “business lunch” version with combined functions and mixed accuracy is about twice as fast on CPUs as the base version. The use of mixed accuracy speeds up the code about 1.6 – 1.7 times and saves memory nearly twice with no effect on accuracy of results, as shown in [12]. The OpenCL version of the code gives significant acceleration, one GPU performs like 7-8 modern multicore CPUs, which is in good agreement with the memory bandwidth ratio between CPU and GPU. On the other hand, it has become more difficult to maintain the code and introduce changes. New modifications must pass through the three versions of code: the full version for CPUs (the so-called “playground”), the “business lunch” reduced version for resource-intensive simulations on CPUs, the OpenCL version for GPUs. However, the good thing is that the amount of the kernel code has been minimized. It fits 5 thousand lines so far, while the full version is about 200 thousand lines. Consistency between the three versions is ensured by an automated QA procedure that includes dozens of tests.

Finally, several scale-resolving applications have proven robustness and performance of the heterogeneous version with the multilevel MPI+OpenMP+OpenCL parallelization.

Acknowledgements. This work has been funded by the Russian Science Foundation, project 19-11-00299. The K60 hybrid cluster of the Collective Usage Centre of KIAM RAS has been used for computations.

REFERENCES

[1] I. Abalakin, P. Bakhvalov, T. Kozubskaya. Edge-based reconstruction schemes for unstructured tetrahedral meshes. *International Journal for Numerical Methods in Fluids* (2016) **81**(6):331–356. DOI: 10.1002/fld.4187

[2] P. Bakhvalov, T. Kozubskaya. EBR-WENO scheme for solving gas dynamics problems with discontinuities on unstructured meshes. *Computers and Fluids* (2017) **157**:312–324.

DOI: 10.1016/j.compfluid.2017.09.004

[3] A. N. Bocharov, N. M. Evstigneev, V. P. Petrovskiy, O. I. Ryabkov, I. O. Teplyakov. Implicit method for the solution of supersonic and hypersonic 3D flow problems with Lower-Upper Symmetric-Gauss-Seidel preconditioner on multiple graphics processing units. *Journal of Computational Physics* (2020) **406:**109189. DOI: 10.1016/j.jcp.2019.109189

[4] P. E.Vincent, F. Witherden, B. Vermeire, J. S. Park, A. Iyer. Towards green aviation with python at petascale. *In: SC16: International conference for high performance computing, networking, storage and analysis. IEEE* (2016) 1–11. DOI: 10.1109/SC.2016.1

[5] R. Borrell, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, O. Lehmkuhl, V. Mehta, H. Owen, M. Vázquez, G. Oyarzun. Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics. *Future Generation Computer Systems* (2020) **107**:31–48. DOI: 10.1016/j.future.2020.01.045

[6] A. Gorobets, S. Soukov, P. Bogdanov. Multilevel parallelization for simulating turbulent flows on most kinds of hybrid supercomputers. *Computers and Fluids* (2018) **173**:171–177. DOI: 10.1016/j.compfluid.2018.03.011

[7] X. Alvarez-Farre, A. Gorobets, F. X. Trias. A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers. *Computers and Fluids* (2021) **214**:104768. DOI: 10.1016/j.compfluid.2020.104768

[8] 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* (2018) **79**(3):786–796. DOI: 10.1016/j.future.2017.09.029

[9] A.Gorobets. Parallel Algorithm of the NOISEtte Code for CFD and CAA Simulations. *Lobachevskii Journal of Mathematics* (2018) **39**(4):524–532. DOI: 10.1134/S1995080218040078

[10] M. Shur, P. Spalart, M. Strelets, and A. Travin. An enhanced version of DES with rapid transition from RANS to LES in separated flows. *Flow, Turbulence and Combustion* (2015) **95**:709–737. DOI: 10.1007/s10494-015-9618-0

[11] A. Gorobets, P. Bakhvalov. Improving Reliability of Supercomputer CFD Codes on Unstructured Meshes. *Supercomputing Frontiers and Innovations* (2019) **6**:44–56. DOI: 10.14529/jsfi190403

[12] A. Gorobets, P. Bakhvalov, A. Duben, P. Rodionov. Acceleration of NOISEtte Code for Scale-resolving Supercomputer Simulations of Turbulent Flows. *Lobachevskii Journal of Mathematics* (2020) **41**(8):1463–1474. DOI: 10.1134/S1995080220080077

[13] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics* (1981) **43**:357–372. DOI: 10.1016/0021-9991(81)90128-5

[14] E. Turkel. Preconditioning Techniques in Computational Fluid Dynamics. *Annual Review of Fluid Mechanics* (1999) **31**:385-416. DOI: 10.1146/annurev.fluid.31.1.385

[15] Van der Vorst H.A. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing* (1992) **13:**631–644. DOI: 10.1137/0913035

[16] F. X. Trias, D. Folch, A. Gorobets, A. Oliva. Building proper invariants for eddy-viscosity subgrid-scale models. *Physics of Fluids* (2015) **27**:065103. DOI: 10.1063/1.4921817

[17] F. X. Trias, A. Gorobets, A. Oliva. A new subgrid characteristic length for large-eddy simulation. *Physics of Fluids* (2017) **29**:115109. DOI: 10.1063/1.5012546

[18] S. Stotz, Y. Guendogdu, R. Niehuis. Experimental Investigation of Pressure Side Flow Separation on the T106C Airfoil at High Suction Side Incidence Flow. *Journal of Turbomachinery* (2017) **139**(5):051007. DOI: 10.1115/1.4035210

[19] M. L. Shur, P. R. Spalart, M. K. Strelets, A. K. Travin. Synthetic Turbulence Generators for RANS-LES Interfaces in Zonal Simulations of Aerodynamic and Aeroacoustic Problems. *Flow, Turbulence and Combustion* (2014) **93**:63–92. DOI: 10.1007/s10494-014-9534-8

[20] A. P. Duben, T. K. Kozubskaya. Evaluation of Quasi-One-Dimensional Unstructured Method for Jet Noise Prediction. *AIAA Journal* (2019) **57**(12):5142–5155. DOI: 10.2514/1.J058162