Performances of the CFD open-source HPC platform TRUST on GPUs

Elie Saikali, Adrien Bruneton, Pierre Ledac

EPJ Web of Conferences **302**, 03004 (2024) https://doi.org/10.1051/epjconf/202430203004
SNA + MC 2024

Performances of the CFD open-source HPC platform TRUST on GPUs

Elie Saikali[1,*], *Adrien* Bruneton[1,**], and *Pierre* Ledac[1,***]

[1]Université Paris-Saclay, CEA, Service de Génie Logiciel pour la Simulation, 91191, Gif-sur-Yvette, France.

Abstract. TRUST is a versatile open-source CFD tool developed by the CEA since 1993. Initially designed for nuclear applications, TRUST has evolved to tackle a range of thermohydraulic challenges, from one-phase to multi-phase flows. It offers various numerical methods and supports different mesh types for efficient computation on diverse computing platforms, including high-performance computers. Recently, efforts have been made to integrate GPU computing libraries like AmgX, rocALUTION, and Kokkos aiming for a hybrid CPU/GPU code achieving better performance portability. This paper provides an overview of the TRUST platform, discusses its GPU computing strategy, and presents selected associated results.

1 Introduction

TRio-**U** **S**oftware for **T**hermal-hydraulics (TRUST) is an open-source (BSD license) software tool for computaional fluid dynamics (CFD). Developed by the Energy Division (DES) of the French Atomic and Alternative Energy Commission (CEA) since 1993 [1], it is built on an object-oriented, parallel approach using the C++ language.

The platform, with its main open-source derived application TrioCFD [2], handles various thermohydraulic challenges, from turbulent one-phase flows to turbulent multi-phase compressible flows. Simulating multi-species Low Mach Number flows (combustion-like model) is also possible, thanks to the weakly-compressible model of TRUST. Initially focused on nuclear applications [3], TRUST is now used in diverse fields like hydrogen safety, Li-ion battery simulations, and PEMFC fuel cells [4, 5].

TRUST offers different numerical methods for various mesh types, supporting finite difference, finite volume, and finite element techniques. It is designed to work efficiently on different computers, including high-performance ones. Simplification and reusability are key aspects. Figure 1 illustrates some mesh types that can be read and used by TRUST to run a dedicated calculation.

Using the METIS library, TRUST enables High-Performance Computing (HPC) simulations by dividing the computational domain into overlapping sub-domains. METIS ensures balanced load distribution among MPI processors thus enhancing performance. In practice,

*e-mail: elie.saikali@cea.fr
**e-mail: adrien.bruneton@cea.fr
***e-mail: pierre.ledac@cea.fr

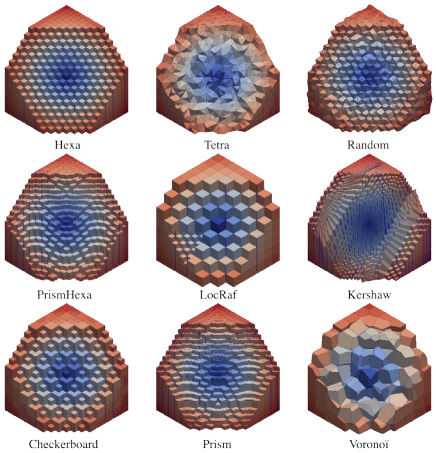

Figure 1. Series of meshes that can be used by the TRUST software.

all sub-domains are normally distributed homogeneously among the processors, which by using MPI communicate only with required neighbor processors when data transfer is needed. Linked with powerful libraries such as PETSc and MEDCoupling, TRUST ensures high performance and scalability.

The software's I/O processes are parallelized, allowing reading and writing from single or distributed files using the HDF5 library. CFD General Notation System (CGNS) [6] is one of the parallel format (based on HDF5) available for post-processing.

TRUST handles large-scale simulations, supporting 64-bit integer representation for domains exceeding 250M cells. Notably, the largest TRUST simulation run to date comprised 2 billion cells distributed over 50K MPI processors [7].

As with all previous historical in-house codes, TRUST was entirely CPU-based. In the coming years however, the power of supercomputers will predominantly come from the compute partition equipped with GPU accelerators, while a minor portion (30 PFlops, at best 3%) of the power will come from traditional scalar nodes. This presents a significant challenge for code developers looking to leverage this new architecture, requiring substantial efforts in porting and algorithmic rewriting.

For this reason, the TRUST platform has been pursuing a strategy of gradually incorporating dedicated GPU computing libraries for certain intensive computational tasks over the past few years. Its goal is to achieve a hybrid CPU/GPU code, ensuring performance portability regardless of the type of partition used on the supercomputer.

In 2020, the Nvidia GPU-accelerated sparse linear algebra library, AmgX [8], was successfully evaluated and subsequently integrated into the platform the following year. In 2022, an equivalent library for AMD GPUs, rocALUTION [9], was integrated into the code as part of a GENCI support contract during the installation of the Adastra supercomputer at CINES [10]. While the results obtained were satisfactory in some aspects, they revealed a lag compared to AmgX, prompting consideration for other solutions. Currently, we are working on supporting multiple architectures (including Nvidia, AMD, and Intel GPUs).

The paper is organized as follows: we will first review the overall architecture of the platform, then the strategy used to port the code to GPU will be detailed. The fourth section will present the most important results obtained so far, and we will finally conclude.

2 TRUST architecture

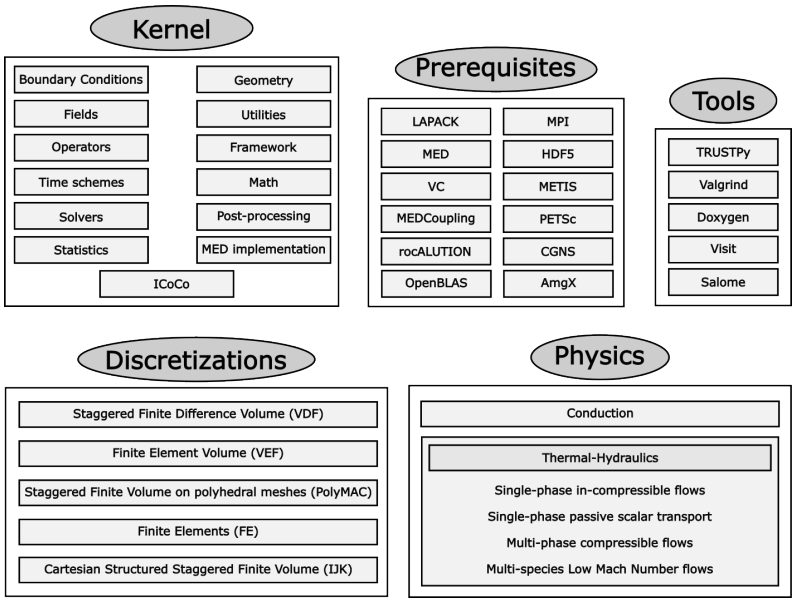

Figure 2. Main TRUST modules and tools.

TRUST is designed with an architecture that ensures robust and efficient computational tasks, thanks to the power of the C++ language and object-oriented principles. The choice of C++ is historical and justified by the following points: 1) it is an object-oriented language that supports inheritance, polymorphism and encapsulation, 2) the language uses strong typing which contributes to achieving reliability objectives, 3) it is a compiled language and is therefore compatible with the level of performance required, 4) it is a de facto standard, with a large installed base. This contributes to the life expectancy of the software and enables it to benefit from the development of a large number of software engineering tools and component libraries.

Modernization using C++11 features like templates, smart pointers, SFINAE (Substitution Failure Is Not An Error [11]), and STL enhances readability and maintainability of the platform. With a focus on simplicity and directness, TRUST emphasizes coarse-grained class inheritance and straightforward loops. Supported by user-friendly tools like Doxygen (for code documentation) and Jupyter notebooks (for validation forms), TRUST provides a modern environment for development, computational exploration, validation/verification and physical analysis.

Figure 2 depicts the main modules (kernel, spatial discretizations and physics), prerequisites and tools of TRUST. The modular architecture of the code makes it easy to add new functionalities, classes and physics. Developers who choose to introduce a new class by specializing an existing one are guided by the architecture, which provides a model to follow. We emphasize that the added classes can be either generic or specific by spatial discetization. Recall figure 1 that illustrates all possible meshes that the code can deal with.

Finally TRUST can be used as a standalone generic simulation software, but can also be extended by building a specialized application re-using part of the modules, or overriding some of the C++ classes found in the code base. Such an application (called *BALTIK*

internally) allows for example to specialize the TRUST functionalities for low scale turbulent multi-phase simulations (TrioCFD [2]), component-scale simulations (e.g. 3D module of CATHARE [12]), or even non-nuclear applications, like batteries and fuel-cell (PEMFC) simulations (and many other CEA internal codes).

3 Porting TRUST to GPU

The total computing cost of a TRUST simulation can be split into two main parts: first the resolution of one (or several) sparse system(s) of the form Ax=B, typically required for obtaining the pressure field when solving the incompressible Navier-Stokes equations, or when impliciting part of the computation (e.g. diffusion implicitation), and second, the actual computation of various physical data, be it the fluxes at the element faces, or the coefficients of the matrix used for an implicit computation for example.

The first part is by far and large the most costly, as it can reach up to 80% of the total computation time [7]. This hence represents the first priority in terms of performance optimization. When running on CPU only, TRUST traditionally uses the PETSc sparse linear algebra library [13]. Although the PETSc project is making an increasing effort to port their library to GPU, at the time of writing this was not mature enough, and we did not investigate further this possibility. In recent years several sparse linear solvers adapted to GPUs have been proposed. Among these we tested in this paper: AmgX [8], rocALUTION [9] and AMGCL [14].

For the second part, OpenMP [15] was formerly used in the TRUST code to port to GPU some of the most intensive compute kernels, but an ongoing effort is done to move to the Kokkos library [16] which allows to write performance-portable code for various types of architectures.

3.1 Solvers

3.1.1 AmgX

AmgX is a high-performance parallel computing, open-source library developed by Nvidia [8]. Designed specifically to accelerate numerical simulations using Algebraic Multi-Grid (AMG) Methods on NVIDIA GPUs, AmgX offers optimized implementations for this architecture. The solvers available in AmgX are as follows, with the first two being the most commonly used in TRUST: CG (Conjugate Gradient), GMRES (Generalized Minimal Residual), BiCGSTAB (BIConjugate Gradient STABilized). Various preconditioners (of paramount importance for an efficient convergence and scalability of the solver) are also available: C-AMG (Classical AMG, Ruge-Steuben) and Unsmoothed Aggregation AMG. Finally AmgX also provides smoothers such as: Jacobi, Gauss-Seidel, SOR, ILU, Chebyshev polynomial type, etc.

The Solv_AMGX class in the TRUST solver hierarchy interfaces with the library AmgX. As can be see on figure 3, the class currently inherits from the Solv_Petsc which implements the calls to the PETSc library. Indeed, the TRUST matrix is stored in PETSc format “mataij”, then transformed into AmgX format via the functions of the AmgXWrapper project [17]. This extra layer deals with the case where the number M of MPI ranks is greater than the number G of GPUs available on the accelerated node. In this case, an agglomeration of M matrices/vectors in G matrices/vectors is made, in order to guarantee (for performance reasons) that each GPU only interacts with a single MPI rank.

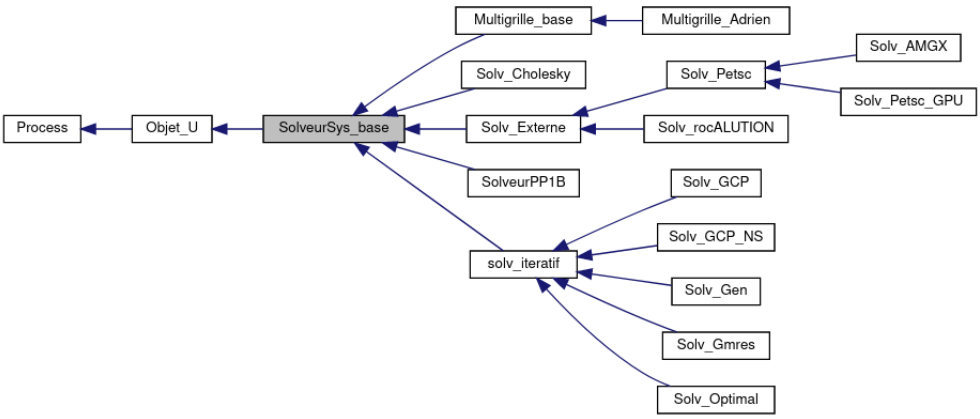

Figure 3. Implementation of the AmgX solver as a TRUST solver

3.1.2 rocALUTION

rocALUTION [9] is an open source numerical computation library optimized for AMD GPUs. Mainly focused on linear algebra, rocALUTION offers features such as factorization, solving linear systems and manipulating sparse matrices. rocALUTION provides a user-friendly, well-documented interface which facilitates its usage.

In TRUST the Solv_rocALUTION class inherits from the Solv_Externe class (see figure 3 again) which brings together information and shared methods for interfacing with external libraries (notably PETSc, AmgX and rocALUTION). This relates in particular to the numbering scheme of matrices and distributed vectors (local, global indices), the management of common items (shared by at least two MPI ranks, such as joint vertices or faces), and conversion methods to the Morse format (CSR).

3.1.3 AMGCL

AMGCL (Algebraic MultiGrid Computation Library [14]) is a C++ library designed for scalable algebraic multi-grid methods, and presented as being efficient on both CPU and GPU. AMGCL supports different types of grids and can be used with various linear algebra libraries notably the CPU based ones (e.g. Intel MKL, Eigen, Boost) and the GPU-based ones (e.g. CUDA). A key point is that AMGCL can be built using the portable OpenCL standard and can thus run on Nvidia GPUs (with Cuda or OpenCL) as well as AMD GPUs (with OpenCL).

In this paper, AMGCL was tested to a lesser extent: no specific class was created in TRUST for its usage. Only the raw solving performance was tested, using a common binary format for the matrix loading [18], and the binary given as an example in the AMGCL project.

3.2 Computation kernels

As mentioned previously, computation kernels used to evaluate various physical data or coefficients represent the second most costly part in a typical TRUST simulation, after the solving of the sparse linear systems.

Contrary to the solver, which is called at a well defined location in the source code, those computations are spread out in many different places in the code and require a much lengthier and tedious migration effort. Those are typically code pieces where we loop on a given geometrical entity (e.g. faces) to compute a given data (e.g. a diffusion flux).

For the last three years the effort was put on OpenMP [15] and its *omp target* pragma directive, but an increasing effort is now being made to port those kernels to use the Kokkos library [16]. Kokkos is being developped as part of the ECP (Exascale Computing Project) in US and allows the writing of C++ calculation kernels that can be executed on CPU or GPU, with the aim of ensuring performance portability on each of the target architectures. Ideally the code should not be re-written when switching architecture and performance should remain. To this end, Kokkos implements a reference abstraction layer to abstract hardware heterogeneity (x86 or Arm processor, Nvidia, AMD, Intel accelerators, etc.) and offers efficient memory access patterns (automatic selection of the data layout, etc.). It provides methods to facilitate execution on machines having several accelerators of different types, but also the management of data between different types of memories like those that are now accessible on modern machines (HBM, DRAM, NVRAM...).

Below we present a concrete example of a kernel used in the divergence operator of TRUST, for the velocity field on a non-structured mesh, using the Finite Element Volume (VEF) spatial discretization. The three listings present the CPU version, the parallelized version using OpenMP, and finally the Kokkos implementation.

```
for (int elem = 0; elem < nb_elem; elem++) {
    double pscf = 0;
    for (int indice = 0; indice < nfe; indice++)  {
        const int face = elem_faces(elem, indice);
        const int signe = elem == face_voisins(face, 0) ? 1 : -1;
        for (int comp = 0; comp < dim; comp++)
            pscf += signe * vit(face, comp) * face_norm(face, comp);
    }
    div(elem, 0) += pscf;
}
```

The OpenMP kernel forces us to explicitely compute the indices expansion when accessing the multi-column arrays, which has proven to be very error-prone, and hard to debug as no bound check is performed:

```
#pragma omp target teams distribute parallel for if (computeOnDevice)
for (int elem = 0; elem < nb_elem; elem++) {
    double pscf = 0;
    for (int indice = 0; indice < nfe; indice++) {
        const int face = elem_faces_addr[elem * nfe + indice];
        const int signe = (elem == face_voisins_addr[face * 2]) ? 1 : -1;
        for (int comp = 0; comp < dim; comp++)
            pscf += signe*vit_a[face*dim+comp]*face_norm_a[face*dim+comp];
    }
    div_addr[elem] += pscf;
}
```

The Kokkos kernel, via its usage of C++ lambda functions, allows to re-use a syntax much closer to the initial CPU one (first listing), while also offering mechanisms to detect potential out-of-bounds indexing:

```
auto kern_ajouter = KOKKOS_LAMBDA(int elem)
{
    double pscf = 0;
    for (int indice = 0; indice < nfe; indice++) {
        const int face = elem_faces_v(elem, indice);
        const int signe = elem == face_voisins_v(face, 0) ? 1 : -1;
```

```
        for (int comp = 0; comp < dim; comp++)
            pscf += signe * vit_v(face, comp) * face_norm_v(face, comp);
    }
    div_v(elem, 0) += pscf;
};
Kokkos::parallel_for("[KOKKOS]Op_Div", nb_elem, kern_ajouter);
```

The memory management mechanism currently still relies on OpenMP, which we manage to associate with the Kokkos view allocations (using so called Kokkos *unmanaged views*). This will be later be changed to rely exclusively on Kokkos functionalities.

Another critical point of attention will be the internal layout of the data in multi-dimensional arrays. Kokkos view mechanism allows to choose between left layout (left-most index varying the fastest) and right layout (right-most index varying the fastest). It is well known [16] that the former provides a better data locality for GPU processing, but the historical set up of TRUST (as all typical CPU applications where cache considerations drives the layout) uses the latter. It is not clear yet whether this should be changed explicitly (performing a deep copy and a transpose of the data), or whether the small extent of the extra dimensions that a typical TRUST multi-dimensional array presents (typically the second dimension of an array is often smaller than 10 in size) should not impact performance, even using the 'wrong' layout.

4 Selected results and discussion

4.1 AmgX

To highlight the performance of AmgX in TRUST on an accelerated partition we give in the table below times and number of iterations when solving systems of respective sizes 2.59, 20.7 and 165.9 million lines (corresponding exactly to the number of pressure unknowns in a tetrahedral mesh for the TRUST VEF P0 spatial discretization) on the CCRT *topaze* supercomputer [19]. The comparison is made with an equal number of compute nodes (0.5, 4, 32) between the scalar partition (128 AMD Milan cores per node) and the accelerated partition (4 Nvidia A100 80GB GPUs per node):

Table 1. AmgX performances on NVidia GPU

	Scalar Partition (PETSc)				**Accelerated Partition (AmgX)**			
Unknowns	CPUs	Tps [s]	Its	Tps/It [ms]	GPUs	Tps [s]	Its	Tps/It [ms]
2,592,000	64	0.270	5	54	2	0.020	5	4
20,736,000	512	0.315	6	52	16	0.033	6	5.5
165,888,000	4096	0.362	6	60	128	0.112	18	6.2

On a small number of nodes AmgX performs better than the PETSc solver (20ms vs 270ms - acceleration factor 13) but it deteriorates to only reach an acceleration factor of 3 on 32 nodes (112ms vs 362ms). The main reason identified, visible in this table, is the lower scalability of multi-grid algorithms on GPU in AmgX (the number of iterations of the its solver increases from 5 to 18) while it is excellent for PETSc (iteration number remaining between 5 and 6).

The two solvers (AmgX and PETSc) are however configured identically: CG and pre-C-AMG conditioning with an identical relative convergence tolerance set at 5.e-4. Only the smoother differs (Jacobi local for AmgX, Chebyshev for Petsc) and can explain the lower scalability of AmgX.

It is clear that one of the challenges of exascale simulations will be the optimization of algorithms and associated parameters to have a scalable convergence rate (stable number of iterations), by having relatively cheap parallel preconditioners.

This being said, the AmgX library (available for 2 years now in TRUST) remains for the moment the reference in terms of performance on GPU for TRUST. Furthermore the constant updates of the upstream project indicates that those scalability issues might be improved in the future.

4.2 rocALUTION

The first tests were carried out on the Adastra supercomputer [10], which was the only platform available to us benefiting from the ROCm environment (equivalent of Cuda for Nvidia). A difficulty was (and remains) to choose the most efficient, scalable GPU preconditioner. Unsurprisingly, multi-grid preconditioners are the most efficient, but we did not succeed to adjust them correctly to have a convergence rate as good as the AmgX one or the PETSc one. In addition this rate degrades very clearly on a weak scaling test (even in a worse fashion than AmgX) as shown by the figures in the table below. The calculations are carried out on the scalar partition (192 AMD Genoa cores per node) and the accelerated one (8 GPUs via 4 AMD MI250X per node):

Table 2. rocALUTION performance on AMD GPU

	Scalar Partition (PETSc)				**Accelerated Partition (rocALUTION)**			
Unknowns	CPUs	Tps [s]	Its	Tps/It [ms]	GPUs	Tps [s]	Its	Tps/It [ms]
2,592,000	64	0.162	5	32	2	0.118	49	2.4
20,736,000	512	0.182	6	30	16	0.316	124	2.5
165,888,000	4096	0.207	6	34	128	0.685	249	2.7

The figures are very disappointing for the classic AMG preconditioner (Ruge Stueben) with a poor convergence rate (10 times more iterations on the first mesh) which does not help performance despite an iteration cost half as much than in AmgX (2.5ms against 5ms). On this first mesh rocALUTION holds a tight win, but then PETSc is systematically faster. According to the documentation, confirmed by discussions with the project support, the majority of rocALUTION multi-grid preconditioners (C-AMG, SA-AMG, UA-AMG) are sequential (in the sense that the grid hierarchy is created on each local subdomain). Only the PW-AMG preconditioner (PairWise AMG) would be parallel, but on the one hand it is not reliable (a bug report was submitted recently following crashes on several MPI ranks) but also very slow: its rate of convergence is completely undermined by the cost of each iteration. The previous table does not therefore contains those results.

4.3 AMGCL

As mentioned before, for AMGCL only pure solving tests using an external matrix format were performed. A pressure matrix on a tetrahedral mesh with Finite Element Volume (VEF) discretization was used and exported from TRUST in the Market Matrix format [18]. This sparse matrix is symmetrical and contains 2,592,000 lines.

A Conjugate Gradient method was chosen and the matrix renumbering option is enabled, which helps reduce the bandwidth, enhancing data locality and minimizing cache misses on the accelerator. In fact these iterative methods are of the *memory bound* type with a limitation on processing speed imposed by GPU memory bandwidth. We compare the performances

obtained with a calculation similar to TRUST in which the AmgX library is enabled with similar options (Conjugate Gradient solver, SA-AMG or C-AMG preconditioner, relative tolerance 5.e-4).

The smoother however differs : Jacobi for AmgX and a SPAI (SParse Approximate Inverse) for AmgCL. A direct LU solver is used for the coarsest grid. The results are found below:

Table 3. AMGCL performance on a TRUST matrix - NVidia GPU

	AmgX		**AMGCL**		
	CG/C-AMG	CG/SA-AMG	CG/C-AMG	CG/SA-AMG	CG/SA-AMG
Grids	8	20	6	4	4
Setup [s]	0.27	0.05	2.33	1.08	1.39
Solve [s]	0.060	0.220	0.075	0.055	0.045
Iterations	7	47	12	12	10

AMGCL performance is at the same level on a single GPU, or even better than AmgX. The lowest AMGCL resolution time (0.045s) is 25% lower than that of AmgX (0.06s). The time spent creating the preconditioner grids is greater with AMGCL (the documentation indicates that the construction of the grids is done on CPU, whereas it is performed on GPU for AmgX) but amortized quickly (done only once when starting a Navier-Stokes simulation of an incompressible fluid). However this can be an obstacle if the simulation needs an update of the Jacobian matrix at each time step: the grid setup time would then be prohibitive. It will then be necessary to find iterative solvers with a less effective preconditioning than multi-grid techniques, but quicker to set up.

In conclusion for AMGCL, scalability on multiple accelerated nodes, with TRUST-like matrices, need further testing to confirm this first good result. The portability of this library on the AMD architecture (recently confirmed to us by the Adastra cluster support team) will also need to be validated.

The quality of the project documentation (tutorials) and the responsiveness of the support must also be stressed as a clear advantage.

4.4 Computation kernels

Four computation kernels were evaluated in TRUST on a representative case, each one of them in the flux computation for a given operator of the Navier-Stokes equation.

Table 4. Performance comparison between Kokkos and OpenMP

Operator	**OpenMP [ms]**	**Kokkos [ms]**	**Gain**
Divergence	2.2	2.2	**0**%
Gradient	3.9	3.7	+**5**%
Diffusion	3.2	2.3	+**28**%
Convection	40.8	54.3	−**33**%

We notice a certain disparity in the performance of the two programming models. The variability of around 30% in one way or the other must be put into perspective: these GPU kernels still run 20 to 30 times faster than their CPU counterpart.

5 Conclusion and prospects

TRUST is a versatile open-source CFD platform that offers various numerical methods and supports different mesh types for efficient computation on diverse computing platforms, in-

cluding high-performance computers. Recently, efforts have been made to integrate GPU computing libraries like AmgX and rocALUTION, aiming for a hybrid CPU/GPU code for better performance portability. We showed that among the various linear solvers that we tested, AmgX (for NVidia) and rocALUTION (for AMD) are the most mature ones, but that AMGCL could be the future reference. The computation kernels can be accelerated using the portable performance library Kokkos, and an on-going effort is made to migrate the current OpenMP kernels to it. Kokkos offers a rich set of features to be assessed in the future like the management of different types of device memory, or the possiblity to have several execution queues (thus increasing the GPU load).

References

[1] CEA-TRUST-Platform, *code website*, `https://cea-trust-platform.github.io/`

[2] P.E. Angeli, U. Bieder, G. Fauchet, *Overview of the TrioCFD code: Main features, VetV procedures and typical applications to nuclear engineering*, in *NURETH 16 - 16th International Topical Meeting on Nuclear Reactor Thermalhydraulics* (2015)

[3] U. Bieder, E. Graffard, Nuclear Engineering and Design **238**, 671 (2008), benchmarking of CFD Codes for Application to Nuclear Reactor Safety

[4] E. Saikali, G. Bernard-Michel, A. Sergent, C. Tenaud, R. Salem, international journal of hydrogen energy **44**, 8856 (2019)

[5] E. Saikali, A. Sergent, Y. Wang, P. Le Quéré, G. Bernard-Michel, C. Tenaud, International Journal of Heat and Mass Transfer **163**, 120470 (2020)

[6] CGNS, *library website*, `https://cgns.github.io/`

[7] E. Saikali, P. Ledac, A. Bruneton, A. Khizar, C. Bourcier, G. Bernard-Michel, E. Adam, D. Houssin-Agbomson, *Numerical modeling of a moderate hydrogen leakage in a typical two-vented fuel cell configuration*, in *International Conference of Hydrogen Safety* (2021)

[8] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh et al., SIAM Journal on Scientific Computing **37**, S602 (2015)

[9] rocALUTION, *library website*, `https://rocm.docs.amd.com/projects/rocALUTION/en/latest/`

[10] AdAstra, *cluster website*, `https://www.cines.fr/calcul/adastra/`

[11] cppreference, *Sfinae*, `https://en.cppreference.com/w/cpp/language/sfinae`

[12] CATHARE, *code website*, `https://cathare.cea.fr/`

[13] S. Balay, S. Abhyankar, M.F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E.M. Constantinescu, L. Dalcin, A. Dener et al., *PETSc Web page*, `https://petsc.org/` (2023), `https://petsc.org/`

[14] D. Demidov, Lobachevskii Journal of Mathematics **40**, 535 (2019)

[15] O. consortium, *standard website*, `https://www.openmp.org/`

[16] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez et al., IEEE Transactions on Parallel and Distributed Systems **33**, 805 (2022)

[17] P.Y. Chuang, L.A. Barba, The Journal of Open Source Software **2** (2017)

[18] N.I. of Standards, Technology, *Matrixmarket*, `http://math.nist.gov/MatrixMarket/index.html`

[19] CCRT, *cluster website*, `https://www-ccrt.cea.fr/`