GPU acceleration of the FINE/FR CFD solver in a heterogeneous environment with OpenACC directives

X.M. Shine Zhai[1], David Gutzwiller[1] Kunal Puri[2], and Charles Hirsch[2]

[1] Numeca-USA, 1044 Larkin St., San Francisco, CA, USA, 94109
[2] Numeca-International, Chaussée de la Hulpe, 189, 1170 Brussels, BELGIUM
{xiaomeng.zhai, david.gutzwiller, kunal.puri, charles.hirsch}@numeca.be

Abstract. OpenACC has been highly successful in adapting legacy CPU-only applications for modern heterogeneous computing environments equipped with GPUs, as demonstrated by many projects as well as our previous experience. In this work, OpenACC is leveraged to transform another Computational Fluid Dynamics (CFD) high order solver FINE/FR to be GPU-eligible. On the Summit supercomputer, impressive GPU speedup ranging from 6X to 80X has been achieved using up to 12,288 GPUs. Techniques critical to achieving good speedup include aggressive reduction of data transfers between CPUs and GPUs, and optimizations targeted at improving exposed parallelism to GPUs. We have demonstrated that OpenACC offers an efficient, portable and easily-maintainable approach to achieve fast turnaround time for high-fidelity industrial simulations.

Keywords: Heterogeneous computing · OpenACC · Computational Fluid Dynamics

1 Introduction

Heterogeneous architectures that encompass both CPUs and accelerators have become increasingly popular in the HPC community. One decade ago in 2010, only 9 supercomputers in the Top 500 list were equipped with accelerators, but the number has since grown fast, reaching 90 in 2015 and 144 in the latest June 2020 Top 500 list [1]. While various accelerators exist to accommodate different needs, such as Graphical Processing Unit (GPU), Intel Xeon Phi coprocessor and Tensor Processing Unit (TPU) etc., the best performing supercomputers tend to rely heavily on GPUs. In fact, it is the computing power from GPUs that makes exa-scale computing within reach in a manner that is economically viable and energy friendly.

To take advantage of GPUs, it is inevitable to adapt existing CPU-only applications. Since most legacy applications have been developed for a long time with rich features, it is often not practical to rewrite them in GPU-native languages, such as CUDA. On the other hand, OpenACC serves as a useful tool in porting

the codes to a variety of heterogeneous systems. As a high-level directives-based programming model, OpenACC successfully helped us adapt a computational fluid dynamics (CFD) codes FINE/TURBO (which specializes in turbomachinery simulations) to be GPU-eligible, and on the Titan supercomputer at the Oak Ridge National Lab (ORNL) a 2X+ GPU speedup in time-to-solution has been demonstrated with a real-world example [2].

While a 2X+ GPU speedup was satisfactory in 2015, the growing interest from the CFD community to complete high-fidelity fluid simulations with less turnaround time has called for more aggressive GPU performance. In this paper, we present recent efforts to leverage OpenACC in achieving 6X to 80X GPU speedup on the Summit supercomputer, using up to 12,288 GPUs. In section 2, we give a brief introduction of the CFD flow solver FINE/FR used for GPU adaptation. Then in section 3, we discuss in detail the techniques leading to the favorable GPU performance, including reduction of data transfers between CPU and GPU, and targeted optimizations that increase the degree of exposed parallelism to GPUs. Strong scalability performance on Summit is presented in Section 4 before we conclude the work.

2 The FINE/FR CFD Solver

2.1 Programming model of FINE/FR

Based on the high order flux-reconstruction (FR) method [4], FINE/FR uses compact computational stencils where the dense mathematical calculations are highly parallelizable. Such workload is well suited for GPUs as they offer significantly more hardware threads to carry out computing with high throughput. On the CPU side, FINE/FR uses a distributed-memory parallel MPI programming model, where the unstructured grids employed are statically partitioned via ParMETIS [5]. By formulation, FR method offers a high degree of accuracy in resolving fine-scale motions compared to conventional Reynolds-Averaged Navier-Stokes (RANS) solutions. As demonstrated by Figure 1, the shock wave boundary layer interaction (SWBLI) [3], a phenomenon critical in the study of compressor stall mechanisms, is highly visible in the high order simulation in the form of lambda shocks on the upper blade surface. The bulk of the execution time is spent on the Runge-Kutta iteration loop, which contains multiple calls to BLAS matrix-matrix multiplication routines, and dozens of additional correction and calculation routines. Written in C++11 standard, FINE/FR uses object-oriented programming throughout the code, and in the core solver algorithms templatization is extensively used. Both the polymorphism and templatization pose some challenges to a neat OpenACC implementation (see discussions in Section 5), but good GPU speedup is not negatively affected.

2.2 Considerations for GPU execution

Since FINE/FR is based on a legacy NUMECA framework, it is prohibitively expensive in terms of developer-hours to drastically change the underlying code

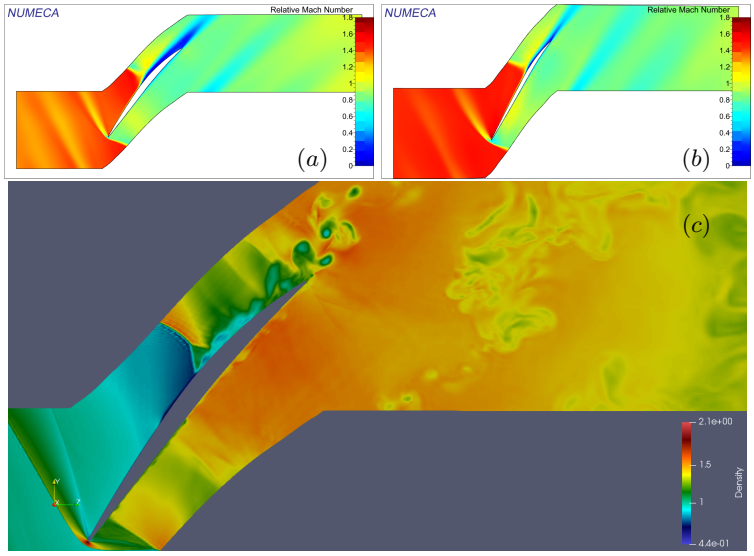

Fig. 1. Relative Mach number at (a) 50% of the blade span and (b) 95% of the blade span using the conventional RANS solution. (c) Instantaneous density snapshot using FINE/FR at polynomial order 4, where shock wave boundary layer interaction (SWBLI) in the form of the lambda-shock can be seen on the upper blade surface

structures. Yet it is useful to list a few considerations for efficient GPU executions of the FINE/FR codes.

- Linearized "flat" arrays favor data transfers: Multiple pointer indirections were natively used in FINE/FR to access array elements, and such usage is supported since OpenACC 2.0. However, current OpenACC implementation transfers each guaranteed-contiguous chunk of memory separately, which can then result in many transfers whose overheads negatively impact the performance. We have replaced multiple pointer indirections by a linearized "flat" array class which stores the data contiguously in memory. Moreover, as explained in Section 3.2, this array class has flags that track the last modified location of the data (CPU or GPU), thus reducing data transfers substantially.
- Sufficient workload and exposed parallelism: As GPUs become more powerful, it is important to saturate GPUs with sufficient workload to obtain significant speedup. While humongous deep learning workloads are a good candidate, we are fortunate that kernels in high-order FR methods usually have plenty of dense math to fully load GPUs too, especially at higher polynomial orders. However, at larger MPI process count, the number of cells/elements in a given partition becomes small and the amount of parallelism exposed to the GPUs is limited. Therefore, GPU speedup inevitably declines at higher MPI process count, and in fact GPU execution may no

longer be cost-effective if the exposed parallelism is too low to outweigh the overheads. Approaches to increase the exposed parallelism are discussed in Section 3.3.

3 Acceleration with OpenACC

Using OpenACC to adapt and accelerate FINE/FR for GPU execution is a natural choice because it does not require a rewriting of the solver in a low-level GPU language, and we had positive experience in adapting another legacy flow solver with OpenACC before [2]. Moreover, OpenACC offers good portability which allows us to conduct rapid code development on local workstations with Intel X86 architectures, and then directly ship the codes to the Summit supercomputer with IBM POWER architecture for scalability tests and production runs. Table 1 shows the system specifications used in this study. In this section, we show various optimizations with performance timed on the local workstation, and in Section 4, scalability performance on Summit is presented.

Table 1. System specifications for OpenACC development and large-scale testing

System	Local workstation	Summit supercomputer
CPU/Host	8 core AMD EPYC	42 core IBM POWER9 node
GPU/Device	1 Nvidia P6000	6 Nvidia V100 per node
PGI compiler	19.9	19.9
MPI library	OpenMPI 2.1.6	Spectrum MPI, 10.2.1.2

3.1 Incremental acceleration of the most time-consuming routines

Generally, it is intuitive to identify the most time-consuming routines and look for opportunities for GPU acceleration. Figure 2 shows a representative stack trace of FINE/FR, using the low-overhead, sample-based profiler HPCToolkit [6]. As noted before, the time marching loop, composed of successive Runge-Kutta iteration steps, constitutes the bulk computation time. The most time-consuming routines were found to be BLAS calls for matrix-matrix multiplication and some thread-safe user routines, both of which are amenable to GPU acceleration via OpenACC. As a first step of acceleration, the following were implemented:

- Replace all BLAS calls with CuBLAS, the CUDA counterpart. Similar changes can be made for non-Nvidia architectures, such as AMD GPUs.
- Instrument remaining user routines (3D loops) with OpenACC pragmas for parallel execution. Minor code changes were necessary to avoid race conditions, for example, by using private variables properly.

– Offload static data such as coordinates and constants to the GPU persistently at the beginning of the program, so that they are readily available on GPUs whenever needed.
– To ensure correct results, all input and output data for each GPU-eligible routine are forced to synchronize in a conservative manner.

The first set of optimizations leads to a 1.5X GPU speedup on the local workstation, and Figure 3 shows the updated resulting stack trace. It is clear that the time-consuming BLAS calls in the CPU run become negligible after using the CuBLAS counterpart, but numerous data transfers between CPUs and GPUs (i.e. between host and device) now dominate the execution time. In fact the amount of data transfers is excessively high due to the conservative approach of data synchronization, which always ensures that data on the host is the most up-to-date. Such an approach is useful to retain correct results when the code undergoes active development, but incurs huge waste for production runs. One alternative is to tailor-code for a particular application where unnecessary data transfers are eliminated and essential host-device communication is overlapped with computations using asynchronous queues. However the lack of generality prevents tailor-coded executables from handling various industrial settings, and would eventually demand continued investment of development efforts. As a result, a systematic and robust solution to minimize the data transfers with low maintenance cost is necessary.

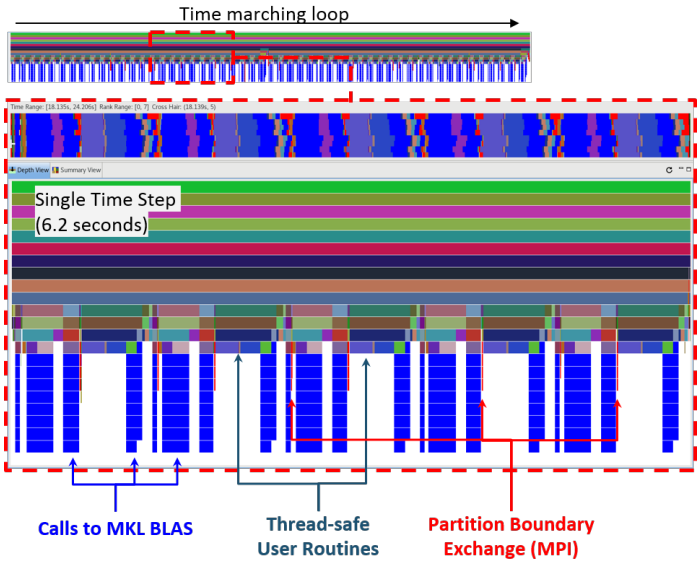

Fig. 2. Stack trace of the CPU-only execution of FINE/FR

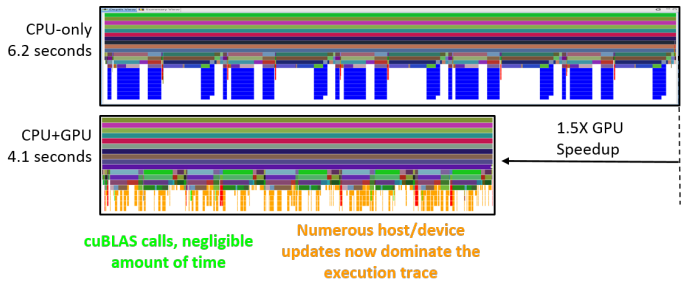

Fig. 3. Stack trace of the CPU+GPU execution of FINE/FR, after the first round of incremental optimization described in Section 3.1

3.2 Minimization of data transfer

The solution to excessive data transfers between host and device is the location-aware arrays. Essentially all major data arrays are wrapped in a container class, which, in addition to holding the linearized array data and host/device update methods, contains a "last modified" flag indicating where the array was last updated. We illustrate the usage in an example as follows.

As frame (a) of Figure 4 shows, the conditional data synchronization between the host and device only occurs when the current access location differs from the saved last accessed location, for the particular array concerned. To ensure coherent data access, in the actual programming shown in frame (b) of Figure 4, it is the developer's responsibility to flag the input and output arrays to GPU-eligible routines, using "sync" and "setLastAccess" calls (see "doubleValues" in the example). In this way, developers can focus on the algorithmic details of a particular routine while data transfers are automatically minimized. For example, the "doubleValues" routine is GPU-eligible but the "addValue" call is forced to run on the host. As a result, data transfers have to occur under the hood. However, should the "addValue" routine be made GPU-eligible, the approach demonstrated would completely avoid data transfers between host and device for the "addValue" call.

Since "sync" and "setLastAccess" calls are prevalent throughout the code, they are referred to as the "GPU boilerplate". The main advantage of "GPU boilerplate" is that through its consistent usage, developers are allowed to follow a "blind incremental acceleration" approach. In other words, as long as the GPU boilerplate is well in place, developers can simply tackle the most time-consuming routines one after another, and an efficient implementation with minimized data transfers would naturally follow. Moreover, new functionalities may be reliably introduced to the host-side codes, with less risk of breaking the data management in a heterogeneous workflow. Yet, it should be stressed that usage of GPU boilerplate must be mandatory for this approach to work. Moreover, existing data structures, especially Arrays of Structures (AoS), can be difficult to retrofit. We

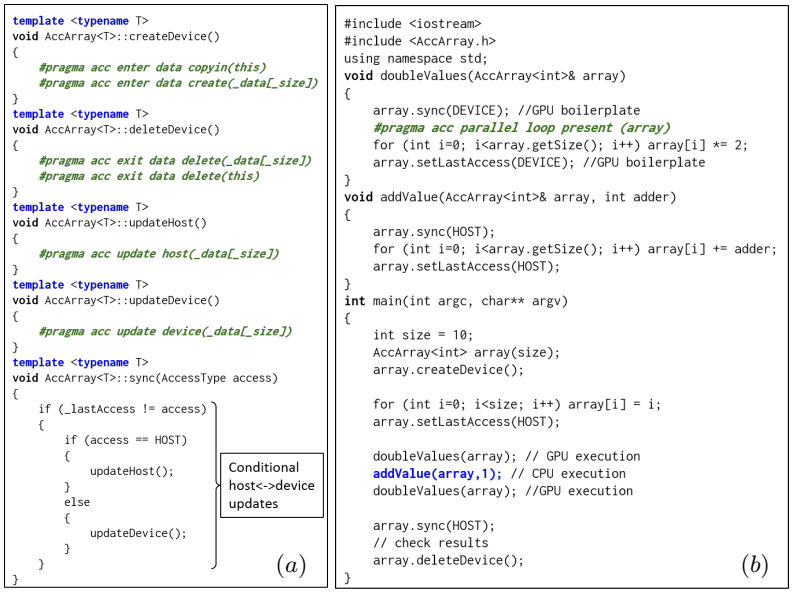

Fig. 4. Code sample demonstrating strategies to minimize data transfer

also note that other established framework exists [7] to automatically mange the issue of data locality.

Performance-wise, Figure 5 confirms that minimized data transfer has significantly improved the GPU performance, leading to a 5.1X speedup compared to the CPU-only execution. In fact, the "GPU boilerplate" has reached a point where all bulk 3D data remain on the device and only 2D data along the partition boundaries needs to be transferred over MPI. Profiling shows that the time to stage the relatively small partition data is too little to warrant further optimization in data transfers, such as GPUDirect and Remote Direct Memory Access. Nevertheless, updated profiling pointed to a handful of routines with poor device performance, which motivated the continued optimization to increase the parallelism exposed to the GPUs as discussed in Section 3.3.

3.3 Optimization to increase exposed parallelism

The high-order FR method routinely goes through a pattern of nested loops, where the outer one loops over element faces and the inner one loops over all the points per face. For unstructured grids, the number of points per face varies depending on the element type and the solution order. For example, Figure 6 shows that the number of flux points is 12 for a quad element, while it is 9 for a triangular element. In the original CPU-only implementation as shown in frame (a) of Figure 7, the code strictly follows the workflow, and a naive

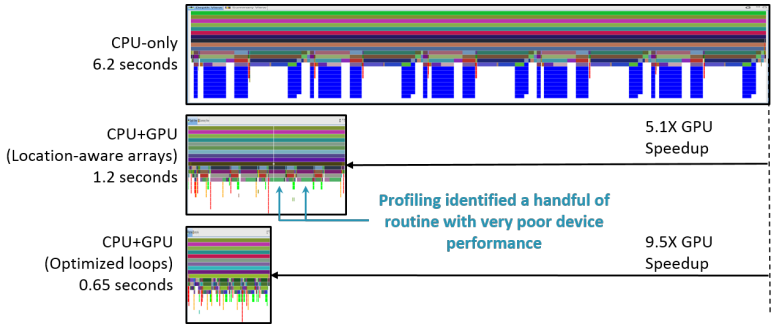

Fig. 5. Comparisons of stack trace in (from top to bottom) CPU-only run; CPU+GPU run, including incremental optimizations and minimized data transfers; CPU+GPU run, fully optimized with increased degree of exposed parallelism

OpenACC adaptation would only parallelize the outer loop, leaving the inner loop sequential thus limiting the degree of parallelism exposed to the GPUs.

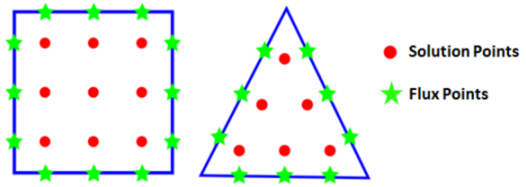

Fig. 6. Demonstration of solution points and flux points on the side of a quad and a trianglular element

Usually when the partition size is sufficiently large, each MPI process contains enough number of faces to saturate the GPUs with computations. However, limited exposed parallelism becomes an issue when the partition size is too small (at a large MPI process count), or when too few elements are treated, such as the boundary data. It turns out that collapsing the loops offers a solution to this problem. Frame (b) of Figure 7 shows the refactored loops in a tightly nested form, where the upperbound of the inner loop is replaced by the maximum value for all iterations. The degree of exposed parallelism can increase by about one order of magnitude at the cost of some threads branching idle. Luckily there is no thread divergence issue involved, and the test on the local workstation showed a 9.5X GPU speedup compared to the CPU-only run (see Figure 5). More detailed tuning of gang/vector parameters for the parallel loops may yield further improved performance, but the optimal parameters are likely problem-dependent. As a result, to avoid reducing portability the default parameters set by the compiler have been used.

```
#pragma acc parallel loop present(....)
for(int iFace=0; iFace<nbFaces; iFace++ )
{
    ...
    int nbPointsFace = getNbPoints(iFace);
    ...
    for (int iPoint=0; iPoint<nbPointsFace; iPoint++)
    {
        <large amounts of thread safe math>
    }
}
```

(a)

```
#pragma acc parallel loop collapse(2) present(....)
for(int iFace=0; iFace<nbFaces; iFace++ )
{
    for(int iPoint=0; iPoint<nbPointsFaceMax; iPoint++)
    {
        int nbPointsFace = getNbPoints(iFace);
        if (iPoint < nbPointsFace)
        {
            <large amounts of thread safe math>
        }
    }
}
```

(b)

Fig. 7. Code refactoring with loop collapse for improved exposed parallelism

4 Scalability of FINE/FR on Summit

The Summit supercomputer represents the state of the art heterogeneous computing architecture in the HPC community, and we are granted access through the INCITE project. Each Summit node contains 42 usable IBM POWER9 CPU cores and 6 Nvidia V100 GPUs, and applications can spawn MPI processes occupying all the 42 CPU cores per node. However as noted before, a large MPI process count gives a small partition size, which may limit the degree of parallelism exposed to the GPUs at scale. Moreover, mapping more than one CPUs per GPU, as managed by CUDA MPS server, may potentially lead to traffic congestion when all data transfers occur at the same time. As a result, Table 2 shows the strong scalability performance of the optimized FINE/FR solver on Summit where one GPU is only paired with one CPU (i.e. NbCPU=NbGPU) to maximize the partition size at large node count. Effectively it means that while GPU runs utilize all 6 GPUs per node in the GPU runs, the CPU runs only use 6 CPUs out of all the available cores as a compromise. A high resolution 8M cells mesh is used in the test with an order 3 polynomial flux reconstruction, and the effective degree of freedom is $8 \times 10^6 \times 4^3 \approx 5 \times 10^8$.

Table 2. Strong scalability using a 8×10^6 cells mesh at order 3 (5×10^8 DoF)

NbNodes	NbCPU & NbGPU	Time (s) CPU	Time (s) CPU+GPU	GPU speedup	NbCell/ Partition	NbDoF/ Partition
8	48	226.00	2.75	82.18	166667	10666667
16	96	117.56	1.54	76.34	83333	5333333
32	192	59.78	0.86	69.51	41667	2666667
64	384	30.59	0.58	52.74	20833	1333333
128	768	15.94	0.47	33.91	10417	666667
256	1536	7.76	0.29	26.76	5208	333333
512	3072	3.57	0.17	20.99	2604	166667
1024	6144	2.15	0.18	11.94	1302	83333
2048	12288	1.00	0.16	6.25	651	41667

The near linear scalability for the CPU-only run through the sweep reflects a well-parallelized and streamlined CPU implementation of FINE/FR and the underlying framework. The GPU implementation attains impressive speedup ranging from 6X+ to 80X+, which not only confirms that performance tuned on the local workstation is easily portable to another system, but also demonstrates the quality of the GPU optimizations. However the GPU runs see a gradual deviation from linear scalability accompanied with a reduction of GPU speedup. As shown by the "number of cells per partition" (Nb/Partition) column in Table 2, the substantial decrease of partition size is responsible for the performance loss as the computational intensity becomes too low to saturate the GPUs. Runs with higher orders on larger meshes can increase the amount of math available to the GPUs, and relax the issue. Continued development work of further code refactoring to expose more parallelism to the device is undertaken.

The 80X+ GPU speedup needs to be interpreted with some caution. For CPU runs, it appears the Intel MKL library is highly optimized for BLAS calculations than the math library available on Summit. While for GPU runs, the CuBLAS routines give optimal performance on the Nvidia V100 cards. Therefore, the GPU speedup on Summit may not act as a perfectly fair performance comparison, and it would be interesting to revisit the GPU speedup on another supercomputer equipped with a well-tuned Intel MKL library.

5 Discussions and conclusions

In this work, we have demonstrated the successful adaptation of FINE/FR, a Flux-Reconstruction based CFD high order solver for heterogeneous CPU/GPU architectures using OpenACC. A highlight of the present work is the use of location-aware arrays, which tracks the location where the array is last accessed. We showed that by consistently adding the "GPU boilerplate", the developers could worry less about the data synchronization between CPUs and GPUs, and focus more on introducing new features and "blindly" optimizing existing bottlenecks one by one. We also showed that increasing the exposed parallelism to GPUs added a 2X boost in parallel performance. It is encouraging to note the 9.5X GPU speedup obtained from incremental optimizations on the local workstation seamlessly translates to impressive speedup on the state of the art supercomputer, which has different CPU architectures and GPU cards, thus demonstrating the nice performance portability of OpenACC. At scale FINE/FR computations using 48 to 12,288 GPUs show favorable speedup in the range of 80X to 6X, and we stress that sufficient computation capable of saturating the GPUs is key to achieving superior GPU performance. It is worth noting that only one version of codes and executable is maintained, and overall the CPU-only execution sees neglible performance impact by the optimization.

OpenACC and the supporting PGI compiler remain actively evolving technologies, and occasionally we have to work around features that are currently not supported. For example, virtual functions and vectors in C++ had to be replaced by non-virtual ones and C-style arrays, and somewhat duplicated codes

annotated by OpenACC had to exist for templated classes. Ease of use will certainly improve as OpenACC becomes more mature.

6 Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DEAC05-00OR22725. The authors are grateful for the comments from the reviewers which have refined the presentation.

References

1. Top 500 list supercomputer statistics in June of 2010, 2015 and 2020, https://www.top500.org/statistics/list/. Last accessed 20 Aug 2020
2. D. Gutzwiller, R. Srinivasan and A. Demeulenaere: Acceleration of the FINE/Turbo CFD solver in a heterogeneous environment with OpenACC directives. In Proceedings of the Second Workshop on Accelerator Programming using Directives, Austin, TX, 2015.
3. E. Touber, and N. D. Sandham: Large-eddy simulation of low-frequency unsteadiness in a turbulent shock-induced separation bubble. Theoretical and Computational Fluid Dynamics, vol. 23, no. 2, pp. 79-107, 2009.
4. A flux reconstruction approach to high-order schemes including discontinuous Galerkin methods. In 18th AIAA Computational Fluid Dynamics Conference, Miami, FL, 2007.
5. G. Karypis, K. Schloegel and V. Kumar: Parmetis: Parallel graph partitioning and sparse matrix ordering library, Dept. of Computer Science, University of Minnesota, 1997.
6. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey and N. R. Tallent: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience, vol. 22, no. 6, pp. 685-701, 2010.
7. M. Ghane, S. Chandrasekaran and M. Cheung: Gecko: Hierarchical Distributed View of Heterogeneous Shared Memory Architectures. Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, pp. 21–30, 2019.