The 8th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2022
5-–9 June 2022, Oslo, Norway

PREPARING A FORTRAN LEGACY CODE FOR THE UPCOMING EXASCALE ARCHITECTURES

JOEFFREY LEGAUX[1] AND GABRIEL STAFFELBACH[2]

[1] CERFACS, Toulouse - France, joeffrey.legaux@cerfacs.fr

[2] CERFACS, Toulouse - France, gabriel.staffelbach@cerfacs.fr

Key words: Computational Fluid Dynamics, High Performance Computing, GPU, ARM

Abstract. Preparing legacy codes for the upcoming exascale systems is a timely topic since the unveiling of the Frontier system in June 2022. In this work we describe the steps taken to prepare the AVBP code for this new step in computing ressources. AVBP [6] is a parallel CFD code that solves the three-dimensional compressible Navier-Stokes equations on unstructured and hybrid grids. AVBP is a cutting-edge software when it comes to distributed memory CPUs, scaling efficiently up to 200.000's of cores on Bluegene or AMD Epyc2 systems. However, other types of architectures such as ARM processors and accelerators are gaining popularity and play a significant role in the exascale era. We first explore the usage of ARM processors, then GPU accelerators through OpenACC[2] directives. This work highlights the difficulties of porting a legacy code to those architectures and solutions implemenented so far for performance.

1 INTRODUCTION

AVBP is a parallel CFD code that solves the three-dimensional compressible Navier-Stokes equations on unstructured and hybrid grids developped by CERFACS[1]. Its highlight is the prediction of unsteady reacting flows in combustor configurations based on the Large Eddy Simulation (LES) approach.

The code is written in Fortran 2003, with some C routines to interface with libraries. I/O is handled via Parallel HDF5 with XDMF descriptors.

AVBP has been essentially developed for x86 many-core CPUs and relies solely on a classical domain decomposition methods over MPI processes for its parallelization, both intra and inter-node. Thanks to careful decomposition that efficiently minimizes the boundaries between domains [7] and an efficient implementation, AVBP exhibits efficient scaling with large number of cores (the current best experiment having reached up to 200.000 cores).

The upcoming exascale era sees a rising trend in the usage of more energy and/or cost-efficient architectures such as GPUs and ARM-based processors. Extending AVBP to support these architectures seems mandatory to benefit from leadership class systems on the INCITE (US) or EUROHPC (EU) programs .

[1]http://wwww.cerfacs.fr

However, AVBP is both a legacy Fortran code that started 25 years ago and a very active research project ; this implies a large and quickly evolving code base, whose developers mostly come from the CFD field and have little to no expertise in the HPC domain per se. Porting it to new architectures thus cannot occur at the expense of its accessibility and maintainability, that is why we need to evaluate if new architectures can be efficiently exploited while still relying on an approachable code base .

2 Test cases

To evaluate the port validity and performance on the various architectures details bellow we used three use case :

- Karman street : This a 2d hybrid mesh simulation on 5998 triangles and 14784 quads of the vortex shedding of a cylinder [1].
- Preccinsta burner : This is a 3d full tetrahedra large eddy simulation of a combustion test rig well-known in the combustion community. [5]
- Explosion simulation : This is a 3d full tetrahedra large eddy simulation of an explosion in a confined space [9].
- Industrial combustion chamber : This is a complex 3d full tetrahedra large eddy simulation of a combustion chamber demonstrator used in the FULLEST Prace project [8]

3 ARM port of AVBP

ARM architectures have been rising and falling for a while. In HPC, until the advent of the A64FX processor from Fujitsu its presence was limited but A64FX showed both the power and potential of ARM for exascale. During this work we focus on portability of the Fortran code base and performance evaluation. The target systems for the test were fore mentioned A64FX architecture, as well as the AMPERE Altra (ARM Ltd) and the graviton2 platforms (AWS). Any of these systems is well documented on the web and will not be discussed here. The main differences are core count (64, 80 and 64 respectively) and the availability of SVE instructions.

To port the AVBP code, the main issue we encountered was the availability of a reliable compiler. Indeed, so far ARM architectures either have their own constructor compilers or rely on the GNU suite. Since GNU is part of the supported compilers for the code, no compatibility issue was discovered and porting was similar as on a basic linux system. However, the compiler can have a major impact on performance. For exemple, when running the karman street test case on a single core of A64FX using either gcc 11 and the Fujitsu compiler, a factor 3 in performance in favor of the constructor compiler was observed without any change on the source code. This is of course understandable since the constructor can tweak compiler behavior knowing the intricacies of the architecture but will impact performance evaluation. Performance on a single core of A64FX remain low when compared to a Intel Xeon Gold 6140 CPU @ 2.30GHz where a Fujitsu core (using the Fujitsu compiler) is 0.3 times the equivalent of the Intel one (using intel parallel studio v19 and ax2 instructions). This is due to the low vectorisation observed in the generated code since the AVBP source code has been thoroughly optimised for Intel x86.

For the remainder of the ARM evaluation, the gcc compiler was used on ARM, Intel and AMD systems.

Figure 1 show the strong scaling efficiency between the different architectures. The references match the names of the instances on the AWS catalog and include Graviton2 (ARM), Intel Cascade Lake, Intel Ice Lake , AMD Rome and Ampere Ultra[4].

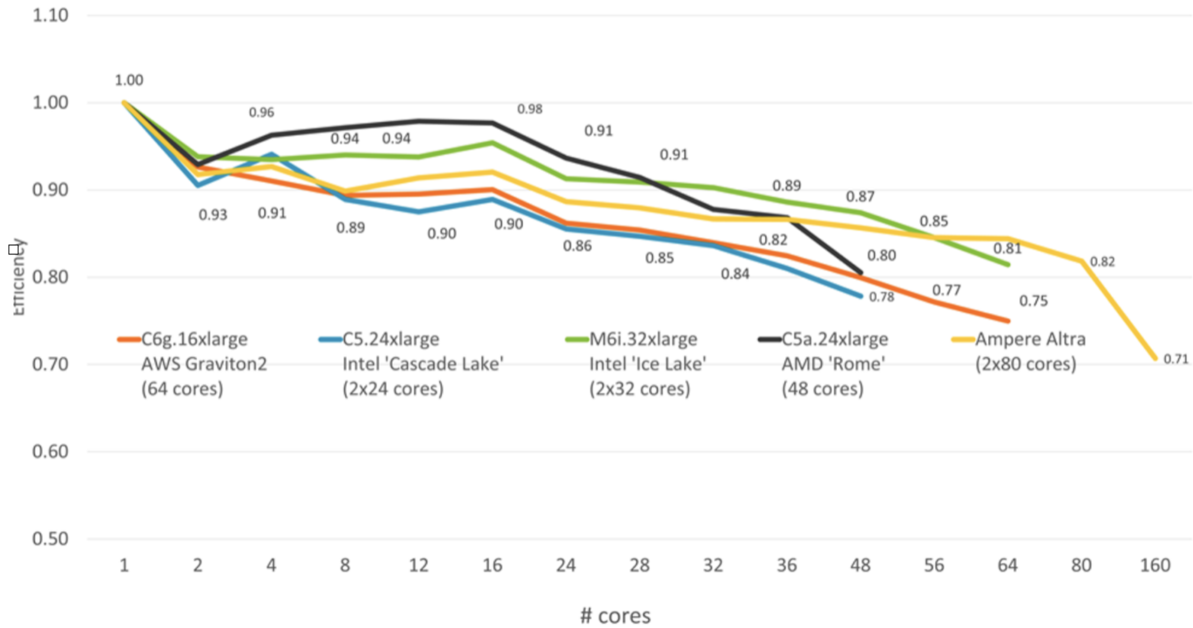

Figure 1: Strong scaling efficiency per architecture of the explosion use case.

These results show that even if per core performance can be lower due to vectorisation concerns, ARM based systems (Graviton and Ampere here) hold their own with the higher core count per socket and a high scaling efficiency of up to 70% on 160 cores (baseline is single core performance). This is very encouraging for the future of these architectures in HPC and provides a glimpse at what would be a full ARM leadership class system for HPC.

4 GPU port of AVBP

GPUs have fundamentally different hardware and programming models than CPUs, porting a code thus requires a specific implementation. Vendor-specific languages such as Nvidia's CUDA are generally considered the best solution that allows to leverage the most performance, however this requires complete rewriting of the program and limits usage to a specific vendor's hardware (unless a compatibility layer is introduced such a HIP for AMD systems). Having a separate specific implementation is unrealistic as only a handful of HPC-aware developers would have to translate all the work of the dozens of mainstream CFD developers.
Symmetrically, having all the developer base switch to a new language is unrealistic until we can prove that it provides major returns on investment performance-wise. High-level approaches

such as Kokkos or C++ parallel algorithms where the developers only have to express generic parallelism which is then automatically interpreted for various hardware would seem perfect candidates and are currently investigated, but they bear the same constraints in terms of adoption and amount of work when dealing with a legacy code.
Given our constraints, directive-based approaches are the most adequate for a first experience as they allow keeping a unified code base where people not interested in GPU usage can still work with the same code as usual and ignore the pragmas.

The two major directive programming models for GPU when this work was started were OpenMP[3] and OpenACC. We chose to use OpenACC since at the time the project started OpenMP was clearly behind in terms of capabilities and compiler support, while having a slightly more complicated syntax. OpenMP 5.0 promises to bridge the gap but is still not fully implemented on any compiler. Additionally, this work benefitted from a close collaboration with NVIDIA, the main supporter of OpenACC.

The porting strategy of the code to GPUs was a complex one as there is not a single obvious compute kernel in AVBP and the execution profile is flat, so we had to try various strategies detailed in the following sections.

4.1 Coarse-grain approach

Most of the load in AVBP comes from computations on structured elements (convection scheme, diffusion scheme, physics models, . . .). The global unstructured mesh is split into small groups of structured elements named *cells*, and those computations always consists at the highest level in a loop over the groups of cells.
Our first approach consisted in coarse-grain parallelism, by parallelizing at the top level on those loops that iterate over the groups of cells. This approach drastically simplifies the number of directives needed, as there are only a handful of those high-level loops.
However the memory management becomes very complicated :

- Arrays that are local to the loop have to be duplicated for each thread that will be running concurrently on the GPU
- Global arrays in modules that are accessed during the loop also have to be identified and moved to GPU memory

Both types of arrays need to be tracked into sometimes deep call stacks, and local automatic arrays have to be declared outside of the top-level loop. This implied heavy and undesirable refactoring of memory allocations. In the end this produced very long GPU threads, and even though we applied it to a simplified workflow extracted from AVBP the performance gain was poor and most importantly produced wrong results ; the long threads acting as a black box on the GPU, it showed to be impossible to debug.

4.2 Fine-grain approach

Having had no success with the coarse-grain approach, we tried its opposite where we would instead focus on porting individual low-level compute-intensive loops, while keeping sequential the high-level loops on the groups of cells. This alleviates most of the difficulties of the porting process : the memory layout remains mostly untouched, every loop can be independently ported, debugged and optimized, and the whole process can be easily splitted among several developers.

This incremental process is relatively straightforward, however the amount of work needed is vastly multiplied. Applying this on the two major computation kernels of AVBP allowed us to emphasize most of the difficulties tied to the code, and we could assess the performance gain on GPU of individual parts of the code. Although those results were individually promising, the global performance of the simulation would actually decrease because of the costly memory exchanges between the main memory and the GPU memory when entering and exiting GPU ported sections. This led us to broaden the scope of our strategy.

4.3 Data-driven approach

The GPU relies on a very fast internal memory to efficiently feed its many cores with their needed data, however transfers between the host main memory and the GPU memory happen at a much slower rate (Nvidia's A100 has an internal bandwidth of 2 TB/s while the typical PCIe 3.0 link only provides around 16 GB/s with the main memory). Experiments with the fine-grain partial port confirmed that large memory exchanges during an iteration represent a major bottleneck.
Therefore we extended the fine-grain approach to a data-centric approach where we put the focus on avoiding as many memory exchanges as possible. This entails the necessity to port non-compute intensive parts of the code that are intuitively less suited for the GPU and expected to be less efficient, but these generally represent very small loads in comparison to efficient intensive loops.
Incrementally applying this approach to the code starting from the previously ported kernels, we ended up rather naturally expanding it to the whole temporal loop that computes all the iterations of the simulation. We managed to push all memory operations on all large data arrays outside of the temporal loop so they would only happen once for the whole duration of the simulation, except from a few unavoidable updates to the main memory when we need to write results to files.
The main drawback of this approach is that we could not have a clear estimate of the global performance until the whole temporal loop would have been ported. Even by capitalizing on the previous work on the fine-grain approach, the whole porting was still a considerable task that took almost a year for 3 full-time developers.

4.4 Limitations

4.4.1 Unified code base

Our main goal was to keep a completely unified code base for the CPU and the GPU implementations, however AVBP has a long history on X86 processors and has been essentially conceived and optimized for these. Eventhough a perfectly unified code base could technically work, there are some parts where the CPU Fortran code could not be efficiently ported only through the addition of OpenACC directives. The performance impact of those unefficient sections was high enough to negate the gains from the rest of the port, thus we had to write specific Fortran parts for the GPU only.

Loops with dependencies Loops where there exists dependencies between iterations cannot be parallelized by the compiler, which ends up generating a sequential GPU kernel. This is critical as this kernel will only use a single core of the GPU, preventing usage of all its other thousands cores. Its execution will be several order of magnitude slower than an efficiently parallelized GPU kernel, and even slower than its execution on a single CPU core.

Such loops were encountered in AVBP in the routines that extracts domain boundaries for inter-nodes exchanges, where indirect indices to access the elements were incremented separately from the loop counter at each iteration. Fortunately, we were able to refactor these routines so that we could express all indices with the loop counter value and a few pre-computed bounds (see listing 1).

Listing 1: Original CPU code

```
lp = 1
dp = 1
DO i=1,runlist_cnt
  cnt = runlist(i)
  SELECT CASE(cnt)
    CASE(1)
      DO j=1,list_length
        lid = indices(lp)
        ofs(1)=(dep_data(dp)...)
        DO k=1,neq
          field_ptr(k,lid)=
            recv_buf(ofs(1)+k)
        END DO
      dp=dp+2
      lp=lp+1
      END DO
    CASE(2)
      ...
      dp=dp+4
```

Listing 1: Code adapted to GPU

```
DO i=1,runlist_cnt
  lp(i) = ...
  dp(i) = ...
END DO
!$ACC PARALLEL LOOP GANG
DO i=1,runlist_cnt
  cnt = runlist_depcnt(i)
  !$ACC LOOP PRIVATE (dp, lp, s, ofs) &
  !$ACC VECTOR(128)
  DO j=1,runlist_length(i)
    dp = runlist_dp(i) + (j-1)*cnt*2
    lp = ...
    DO l = 0, cnt
      ofs(cnt) = dep_data(dp...)
      DO k=1,neq
        DO l=1,cnt
          s(k) = s(k) +
            recv_buf(ofs(l) + k)
          ...
      DO k=1,neq
        !$ACC ATOMIC UPDATE
        field_ptr(k,lid(lp))=s(k)
```

Automatic arrays Automatic arrays in Fortran are arrays that are declared inside a subroutine and whose size depends on the subroutine parameters. This is a convenient feature on CPU as it allows for implicit allocation of those arrays and those are generally – depending

on the compiler and the options used – allocated on the heap which has plenty of available space.
This can easily become problematic on GPU when a subroutine with automatic arrays is called inside a parallel region : GPU threads have a very limited heap space since they are numerous and need to share a limited amount of memory, thus even small automatic arrays can quickly overfill this local thread heap space. Furthermore, memory allocations on the GPU cannot be parallelized. Having a large number of threads sequentially allocating their automatic arrays one at a time can become a major performance bottleneck.
There is no universal solution to address this issue, however a common pattern that is often applicable is to displace the declarations of the automatic arrays outside of the parallel region of the code – turning them into arguments of the subroutine – and declaring them as private in the ACC PARALLEL directive (see listing 2) .

Listing 2: Original code

```
SUBROUTINE correct_face (grid, ...)

!$ACC PARALLEL LOOP ...
DO nf=1,nflen
...
  CALL face_permeable (grid, ...)
...
END SUBROUTINE correct_face

SUBROUTINE face_permeable ( grid, ...)

REAL(pr) :: Yspec_p(1:grid%neqs,1:4)
REAL(pr) :: Yspec_c(1:grid%neqs,1:4)

!$ACC ROUTINE SEQ
...
```

Listing 2: Code without auto arrays

```
SUBROUTINE correct_face (grid, ...)
REAL(pr) :: Yspec_p(1:grid%neqs,1:4)
REAL(pr) :: Yspec_c(1:grid%neqs,1:4)

!$ACC PARALLEL LOOP &
!$ACC PRIVATE(Yspec_p, Yspec_c)
DO nf=1,nflen
...
  CALL face_permeable ( grid, &
   ... , Yspec_p, Yspec_c)
...
END SUBROUTINE correct_face

SUBROUTINE face_permeable ( grid, &
    ... , Yspec_p, Yspec_c)

REAL(pr) :: Yspec_p(1:grid%neqs,1:4)
REAL(pr) :: Yspec_c(1:grid%neqs,1:4)

!$ACC ROUTINE SEQ
...
```

Local optimizations Specific optimizations that aim at reducing the workload on the CPU but sacrifice parallelization potential are generally detrimental on a GPU.
A frequent occurrence in AVBP is when a value is computed in an outer/intermediate loop (typically a value for a whole cell or vertex) and then used over several iterations in an inner loop. This effectively prevents parallelization of the inner loop, displacing the computation into the inner loop proves largely beneficial to the performance on the GPU (see listing 3). Eventhough this "wastes" resources by computing the same value over several threads, it allows to leverage more parallelism by collapsing all the loops which increases the GPU loads, and often exhibits better memory accesses (coalescence).

Listing 3: Original CPU-optimized code

```
!$ACC PARALLEL LOOP COLLAPSE(2)
DO n = 1, nlen
  DO nv = 1, nvert
    denom = one / w(1,nv,n)
    !$ACC LOOP SEQ
    DO k = 1, grid%neqs
      wprim(k,nv,n) =
        w_spec(k,nv,n) * denom
    END DO
  END DO
END DO
```

Listing 3: Code optimized for GPU

```
!$ACC PARALLEL LOOP COLLAPSE(3)
DO n = 1, nlen
  DO nv = 1, nvert
    DO k = 1, grid%neqs
      wprim(k,nv,n) =
        w_spec(k,nv,n) / w(1,nv,n)
    END DO
  END DO
END DO
```

Similarly, manually unrolled loops allow the compiler to generate more efficient CPU code for specific cases but deprive from another level of parallelization, and rewriting them in the form of a generic inner loop may prove beneficial to the performance on GPU.

Loops that are putting pressure on registers Memory resources are relatively scarce on GPUs ; they are similar in global quantity than on a CPU but are distributed over thousands of cores instead of dozens. This leads to a limited amount of memory per GPU core (an A100 has for example an average of 8kB registers and 6 kB L1 cache per core).
Complex loops that carry many computations for each individual iteration might end up using all the available register and cache space available, triggering register spilling to the global memory which is a very time-consuming process. We had to refactor such loops in order to limit register usage by reducing the amount of local variables used or splitting them into several smaller loops.
Another non-intrusive possibility is to limit the OpenACC vector length to less than 32 threads. This will effectively limit the number of concurrent threads inside each Streaming Multiprocessor of the GPU, so that each thread will individually have more resources and this may prevent spilling. This is however a last resort when refactoring cannot solve the issue, since we are effectively reducing the amount of cores used in the GPU.

4.4.2 Workload tweaking

Having an almost common codebase for both CPU and GPU with specific optimized sections still does not automatically guarantees good performance to the end user. Habits inherited from CPU usage need to be reconsidered and architectural particularities have to be taken into account in the parameters of the simulation.

Domain decomposition A typical domain decomposition for CPUs in AVBP consists in a small subdomain for each core of the CPU. GPUs need to have intensive workloads to efficiently feed their thousands cores with computations, so a typical efficient subdomain for a GPU has to be the size of several dozens of CPU subdomains.

Cells groups size The breakdown of the structured computations into groups of cells was originally aimed at optimizing cache memory usage on a CPU. This would be achieved by using relatively small groups of cells, a typical size being 100 cells per group. On the GPU, we want larger workloads to be efficient so this size has to be increased. Benchmarks

have showed that increasing the size is always providing more performance although with diminishing returns. A size of 100.000 usually brings the performance to the right order of magnitude, but ideally one wants to set the group size at the local domain size to maximize the performance.

Constraints on memory usage AVBP is relatively frugal with memory usage, typically requiring around 200MB of memory on a CPU core, and users seldom have to take care of memory consumption. On the other hand, the amount of memory on a GPU is more restrained and becomes a concern when applying the two aforementioned tweaks. Larger domains obviously occupy more memory, but increasing the cells groups size also has a major impact. There are many local arrays that are used for the cells groups computation but they were considered negligible on CPU as the number of cells was small, however when increasing it several thousands time their size quickly reach several GB which puts substantial pressure on the GPU memory.
Since both increasing the subdomain and the cells groups size provide more performance and increase memory usage, the user has to try and find a balance between those two parameters to get the most efficient runs.

GPU binding Typical accelerated nodes on supercomputers contain 4 GPUs, ensuring that the MPI processes are correctly spread across these GPUs inside each node is crucial. Taking care of correct GPU binding is a rather simple task, but represents additional work for the end-user which is used to blindly trust MPI for process placement.

4.4.3 GPU-direct MPI exchanges

During each iteration of AVBP, several inter-process MPI communications happen, using both point-to-point (for domain boundary exchanges) and global (computing the global timestep for instance) patterns.
A classical way to conduct those exchanges would be to copy the concerned data to the CPU from the GPU memory, proceed to the MPI exchanges, then write back the modified data.

Thankfully, OpenACC provides mechanisms to require explicit usage of GPU memory when calling a subroutine. When combined with a *cuda-aware* MPI library, this allows for so-called GPU-direct communications where the MPI transfers can directly exchange data between the memories of GPUs.
Experiments have shown that this mechanism is mandatory to preserve both single-node performance and scalability across multiple nodes with GPUs such as illustrated in figure 2. However its usage depends on the availability of a cuda-aware MPI implementation on the targeted computer.

4.5 Integration and Performance

The GPU full port started based on the AVBP 7.5 release, however when it had been achieved a year later there had already been two major AVBP releases. Additional work was required to catch up, but ultimately GPU support was officially integrated starting from release 7.7.

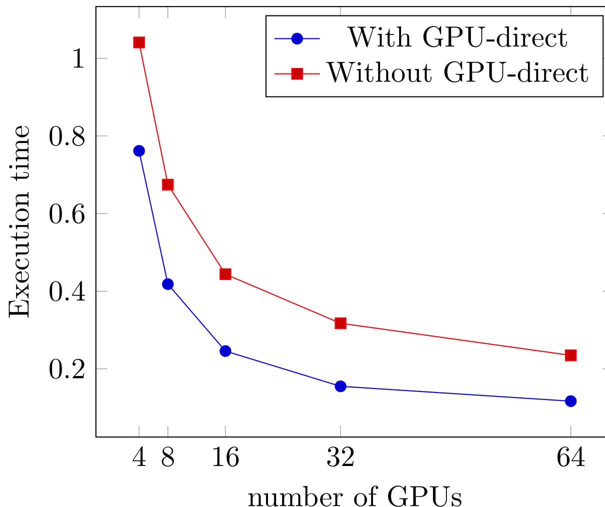

Figure 2: GPU-direct impact on peformance and scalability (sec/iteration on Preccinsta test case)

The first port targeted a handful of representative research test cases that were evaluated on the JEAN-ZAY supercomputer from IDRIS. Table 1 summarize the acceleration factor we obtained when comparing execution speed on a full node either using CPUs only vs using its 4 GPUs.

The port was later extended during an IDRIS Grand Challenge where the goal would be to run a challenging complex simulation. This allowed us to add the necessary support to simulate an industrial combustion chamber model, as well as assessing the performance on a rather large scale (up to 1024 GPUs).

Meanwhile background work carried on to extend the support to the new releases of AVBP and applying optimizations when opportunities were found, thus increasingly leveraging more performance with each new release. We similarly assessed the port performance on JUWELS BOOSTER 2 which has next generation hardware both for its CPUs and GPUs ; the results were coherent with what we observed on JEAN-ZAY.

CPU vs GPU (full node)	AVBP 7.7	AVBP 7.8	AVBP 7.9
Preccinsta	1.82	3.9	4.7
Explosion	1.84	4.2	4.80
Industrial CC	Not supported	3.5	5

Table 1: Acceleration factor on one node of JEAN-ZAY (2x20core Intel Xeon 6248 + 4 V100)

4.6 Conclusion and perspectives

We achieved a GPU port of AVBP that supports a broad selection of complete workflows. This represented a fair amount of work even with the limited amount of code introduced and modified thanks to the directive-based approach. Using GPUs exhibit significant gains in compute time, however those are not high enough to discard pure CPU usage. We also measured a

CPU vs GPU (full node)	AVBP 7.7	AVBP 7.8	AVBP 7.9
Preccinsta	1.9	1.9	2.9
Explosion	2.2	2.3	3.65
Industrial CC	Not supported	Not supported	4.8

Table 2: Acceleration factor on one node of JUWELS BOOSTER (2x24core AMD Epyc 7402 + 4 A100)

significant advantage regarding energy consumption on our cluster, with 40% less energy required to complete a simulation with Nvidia A30 GPU than with Intel Xeon 6140 CPUs.

We originally focused on OpenACC for the GPU port as at the time it was clearly the easiest alternative and there were almost only NVidia GPUs available in supercomputers. However OpenMP has been bridgind the feature gap with its 5.0 version, and new competitors such as Intel and AMD have been recently introducing new GPUs in the HPC landscape that will require OpenMP implementations to use them, so our next major focus will be to convert the existing OpenACC implementation to OpenMP to benefit from a broader range of options in the future.

5 ACKNOWLEDGMENTS

This work has been supported by the EXCELLERAT project which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823691.
The authors would like to thank NVIDIA, the GENCI Cellule de vielle technologique, IDRIS, HPE for their support on the GPU port activities and Amazon Web Services and ARM Ltd for their support on the port to ARM and AWS instances.

REFERENCES

[1] Karman street vortex. `https://hmf.enseeiht.fr/travaux/CD0506/mci/reports/avb1/marl.htm`.

[2] OpenACC homepage. `https://www.openacc.org/`.

[3] OpenMP homepage. `https://www.openmp.org/`.

[4] Running avbp industrial code on arm neoverse n1. `https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/running-avbp-cfd-code-on-arm-neoverse-n1`.

[5] P. Benard, G. Lartigue, V. Moureau, and R. Mercier. Large-eddy simulation of the lean-premixed preccinsta burner with wall heat loss. *Proceedings of the Combustion Institute*, 37(4):5233–5243, 2019.

[6] N Gourdain, L Gicquel, G Staffelbach, O Vermorel, F Duchaine, J-F Boussuge, and T Poinsot. High performance parallel computing of flows in complex geometries: II. applications. *Computational Science and Discovery*, 2(1):015004, nov 2009.

[7] Pavanakumar Mohanamuraly and Gabriel Staffelbach. Hardware locality-aware partitioning and dynamic load-balancing of unstructured meshes for large-scale scientific applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Carlos Pérez Arroyo, Jérôme Dombard, Florent Duchaine, Laurent Gicquel, Benjamin Martin, Nicolas Odier, and Gabriel Staffelbach. Towards the large-eddy simulation of a full engine: Integration of a 360 azimuthal degrees fan, compressor and combustion chamber. part ii: Comparison against stand-alone simulations. *Journal of the Global Power and Propulsion Society*, (May):1–16, 2021.

[9] Pedro S. Volpiani, Thomas Schmitt, Olivier Vermorel, Pierre Quillatre, and Denis Veynante. Large eddy simulation of explosion deflagrating flames using a dynamic wrinkling formulation. *Combustion and Flame*, 186:17–31, 2017.