Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

OpenMP offload at the Exascale using Intel® GPU Max 1550: evaluation of STREAmS compressible solver

Francesco Salvadore
CINECA
Giacomo Rossi
giacomo.rossi@intel.com
Intel Corporation Italia S.p.A
Srikanth Sathyanarayana
Max Planck Computing and Data Facility
Matteo Bernardini
Sapienza University of Rome

Research Article

Keywords: CFD, GPU, OpenMP, Compressible Flows, Ponte Vecchio

Posted Date: April 2nd, 2024

DOI: https://doi.org/10.21203/rs.3.rs-4180620/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

OpenMP offload at the Exascale using Intel® GPU Max 1550: evaluation of STREAmS compressible solver

Francesco Salvadore[1], Giacomo Rossi[2*], Srikanth Sathyanarayana[3], Matteo Bernardini[4]

[1]HPC Department, CINECA, via dei Tizii 6/B, Rome, 00185, Italy.
[2*]Intel Corporation Italia S.p.A., Via Santa Maria Valle 3, Milan, 20123, Italy.
[3]Max Planck Computing and Data Facility, Gießenbachstraße 2, Garching, 85748, Germany.
[4]Department of Mechanical and Aerospace Engineering, Sapienza University of Rome, via Eudossiana 18, Rome, 00184, Italy.

*Corresponding author(s). E-mail(s): giacomo.rossi@intel.com;
Contributing authors: f.salvadore@cineca.it; srcs@mpcdf.mpg.de; matteo.bernardini@uniroma1.it;

Abstract

Nearly 20 years after the birth of general purpose GPU computing, the HPC landscape is now dominated by GPUs. After years of undisputed dominance by NVIDIA, new players have entered the arena in a convincing manner, namely AMD and more recently Intel, whose devices currently power the first two clusters in the Top500 ranking. Unfortunately, code porting is still a major problem, even more so with the presence of different vendors, but at the same time the emergence of simplified standard paradigms suggests an encouraging prospect for developers.

In this work, we analyze the porting and performance of STREAmS, a community code for compressible fluid dynamics, on Intel® Data Center GPU Max 1550 (formerly called Ponte Vecchio or PVC) based architectures. First, we discuss the porting, based on the offload functionality of the OpenMP 5.x paradigm, and in particular using a hybrid directives/APIs approach that fits smoothly into the multi-backend software ecosystem of STREAmS-2. Second, we analyze the performance of the code on two benchmark clusters powered by PVC, including the exascale Aurora cluster. The performance is evaluated at the different levels

of parallelism involved, i.e., the intrinsic parallelism of the PVC tile, the inter-tile parallelism within the GPU configuration, between the GPUs within the node, and between the nodes within the cluster. The analysis shows that although the implementation complexity of the OpenMP porting is limited, it is necessary to follow some important guidelines to achieve satisfactory performance. The PVC GPU shows about 40% higher performance than the NVIDIA A100 or AMD MI250X GPUs, which however were released about 3 years earlier. Both intra-node and inter-node scalability show good results. Overall, the introduction of PVC into the GPU computing HPC landscape represents a positive step forward for diversification and competitiveness in the sector.

Keywords: CFD, GPU, OpenMP, Compressible Flows, Ponte Vecchio

1 Introduction

High Performance Computing (HPC) has made incredible advances over the last two decades, enabling ground-breaking research in simulation, data analysis and artificial intelligence. With the advent of exascale computing in the current decade, Graphical Processing Units (GPUs) have been at the center of much of this progress. For the most part, NVIDIA has dominated the GPU market, but in recent years AMD has emerged as a worthy challenger with its flagship Instinct accelerator series. This is evidenced by the choice of GPU architectures in some of the world's fastest supercomputers [1]. Frontier, currently the fastest supercomputer at Oak Ridge Laboratory, uses AMD GPUs. The two fastest pre-exascale supercomputers in the EU [2], LUMI [3] and Leonardo [4], are powered by AMD and NVIDIA GPUs, respectively. With the introduction of the Ponte Vecchio (PVC) GPU based on the X^e microarchitecture, Intel has also become a serious contender to potentially deliver similar performance to NVIDIA and AMD GPUs. This is evidenced by the establishment of LLNL's Aurora supercomputer, which is the second fastest in the world at the time of this writing, but is expected to be the fastest supercomputer at full capacity, capable of a peak performance of 2 exaFLOPS.

From a hardware perspective, GPUs have come a long way in a short period of time, with newer designs combining CPU and GPU in a unified memory architecture, which is bound to increase overall bandwidth. Unfortunately, programmability and portability are still major issues, with vendor-specific paradigms – particularly NVIDIA CUDA [5] – still dominating the panorama. This is particularly true in the context of Computational Fluid Dynamics (CFD), both for basic research codes [6] and for applications closer to realistic cases [7].

However, portable approaches are beginning to gain significant traction, both as performance portability libraries (e.g., kokkos [8], Raja [9], and alpaka [10]) and as standard paradigms such as OpenMP [11], OpenACC [12], SYCL [13], or the C++ Parallel Standard Template Library [14]. The standard paradigms offer the optimal solution in perspective, but currently their implementation in the various compilers is limited/optimized only for some devices, making portability effective only on

paper. Nevertheless, several studies already demonstrate the potential effectiveness of standard approaches in different fields: for example, SYCL used for biological sequence alignment [15], PSTL for astrophysical applications [16], or OpenACC for CFD applications [17].

From a programming language perspective, Fortran and C/C++ remain the HPC reference languages, and Fortran in particular is widely used in the CFD context for historical reasons, but also for its ability to address the needs of new users to efficiently implement the typical algorithms involved in CFD. However, Fortran's support for heterogeneous computing paradigms is limited in several cases. For example, the performance portability libraries are C++ oriented, and HIP [18], the basic paradigm for programming AMD GPUs, also provides limited support for the Fortran world.

Over the years, several popular Fortran codes have taken different approaches to the portability problem. Neko [19], a spectral element based code, uses multi-level abstractions through the Fortran abstract type with extending implementations for the architecture of choice. GENE [20], a plasma microturbulence simulation code, uses a similar approach with the GPU implementations abstracted using the Gtensor library (developed in C++), which is designed to handle the multi-dimensional arrays similar to Fortran. Quantum ESPRESSO, a suite of codes for electronic-structure calculations and materials modelling, initially started with a CUDA Fortran implementation to target NVIDIA GPUs, but in recent years has moved more towards an OpenACC/OpenMP standard implementation to achieve performance portability [21, 22]. FUN3D, a CFD solver developed at NASA Langley, uses FLUDA, a thin abstraction layer, to target NVIDIA and AMD GPUs. Recently, FLUDA has been extended to handle Intel GPUs via the Intel oneAPI SYCL [23]. Alya, a popular CFD code, has been extended to improve algorithmic scalability using the PSCToolkit [24] and potentially address the exascale scenario.

STREAmS-2 [25, 26] is a state-of-the-art in-house Direct Numerical Simulation (DNS) solver designed to simulate canonical compressible wall-bounded flows (see e.g., [27–29]). It is written in modern object-oriented Fortran and is able to target CPU and GPU architectures, in particular NVIDIA and AMD. Over the years, STREAmS-2 has closely followed and responded to the evolution of GPU technology. The last version of STREAmS-2 supporting NVIDIA and AMD GPUs showed an impressive weak scalability for the EU's top two supercomputers, LUMI and Leonardo [30]. However, with the inevitable arrival of powerful Intel GPUs, an implementation targeting them has been lacking. For Fortran simulation codes like STREAmS-2, OpenMP offloading is a convenient way to efficiently support Intel GPUs. OpenMP has long been the gold standard for thread-based parallel programming models. Since version 4.5, the OpenMP standard has specified support for heterogeneous systems. The key feature of OpenMP is its robust, easy-to-implement benefits, coupled with the continued evolution of the standard. This is promising for codes written in Fortran, as it could potentially be the de facto programming model for targeting all GPUs, helping to achieve true performance portability.

In this paper we present an OpenMP implementation of STREAmS-2. As a first step towards full performance portability, we evaluate its performance on the latest Intel Ponte Vecchio GPUs. At the time of writing, we believe this is the first work to

evaluate the performance of a pure Fortran OpenMP implementation on Intel Ponte Vecchio GPUs. In section two, we begin with a discussion of the capabilities of the Intel Ponte Vecchio GPU, followed by the OpenMP implementation and automation of STREAmS-2. In section three, we discuss the performance results, first on a single tile, followed by the scalability results. Finally, in section four, we report the conclusions of the analysis.

2 Material and methods

2.1 Intel Data Center GPU Max

Intel has a long and succesful story in HPC, with x86 CPUs, from almost three decades: Xeon CPUs, designed for servers and high performance applications, were introduced in June 1998 and have steadily increased the available computing power, also thanks to the introduction of vector instructions (and dedicated hardware) such as SSE, AVX, AVX2 and finally AVX512.

Intel Data Center GPU Max (codenamed Ponte Vecchio) is the first Intel X^e GPU microarchitecture dedicated to High Performance Computing: this new microarchitecture uses both EMIB 2.5D and Foveros packaging technologies to combine up to 128 X^e cores on a single GPU. Unlike previous Intel GPU generations, this microarchitecture uses the X^e core as the compute unit: an X^e core contains vector and matrix ALUs called vector and matrix engines.

The X^e core contains 8 vector and 8 matrix engines (XVE and XMX, respectively) and a large 512KB L1 cache: each vector engine is 512 bit wide and supports 16 FP32 SIMD operations with fused FMAs. With 8 vector engines, the X^e core delivers 512 FP16, 256 FP32, and 128 FP64 operations per cycle. Each matrix engine is 4096 bit wide, so with 8 matrix engines, the X^e core delivers 8192 int8 and 4096 FP16/BF16 operations per cycle; in addition, the X^e core provides 1024B load/store bandwidth to GPU memory.

Each vector engine is a multi-threaded SIMD processor with up to 8 threads available. When a kernel is submitted to the Compute Command Streamer (CCS) for execution, it is divided into work groups. Each work group is divided into vector engine threads that execute it in parallel.

Ponte Vecchio GPU is available in two different models: single tile Intel Data Center GPU Max 1050 and two tiles Intel Data Center GPU Max 1550. In this paper, we discuss the performance of the STREAmS-2 code on the Intel Data Center GPU Max 1550: this PVC model, clocked at 1600 MHz, has 128 GB of HBM2e memory, providing a total bandwidth of 3276.8 GB/s and 128 X^e cores, for a total of 1024 vector engines and 8192 hardware threads. The GPU is organized in two stacks, called tiles, in which the available resources are shared equally; the TDP is 600W and in figure 1 the full dual-tile stack can be observed.

The table 1 shows the peak bandwidth and peak FP64 performance for the Intel GPU Max 1550, along with current GPUs available from Nvidia and AMD. In this paper, we compare PVC performance with Nvidia A100 and AMD MI250X GPUs.

Programming models available for PVC are: OpenCL and Level-Zero at lower level, SYCL/DPC++ for C++ and directive based OpenMP offload for C/C++ and

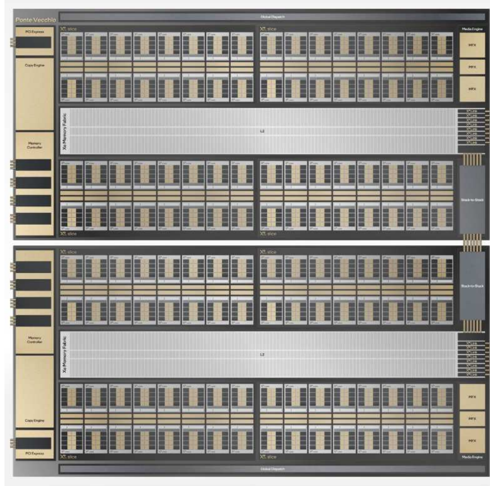


Fig. 1: Intel PVC Stack

Table 1: Memory bandwidth and FP64 theorethical peak performance for currently available GPUs

	PVC	A100 (PCIe)	H100 (PCIe)	MI250X	MI300X
Peak Bandwidth [GB/s]	3276.8	1935	2000	3200	5300
Peak FP64 Performance [TFlop/s]	52	9.7	26	47.9	81.7

Fortran, together with standard programming language parallelization as Fortran `DO CONCURRENT`. The GPU codes must be compiled using Intel oneAPI new compilers (icx, icpx or ifx) and can be linked with Intel oneMKL with GPU offload capability and Intel MPI which supports GPU buffers for GPU to GPU direct communication.

2.2 Numerical code and test case

STREAmS is a CFD code that solves the compressible Navier-Stokes equations in Cartesian coordinates using finite difference discretization. It is capable of simulating both calorically and thermally perfect gases.

The nonlinear terms are evaluated by exploiting a hybrid discretization that switches between central schemes in smooth regions of the flow and shock-capturing schemes for regions with discontinuities. Among the central schemes, skew-symmetric schemes are available, which are cast in terms of numerical fluxes to allow easy hybridization with the shock-capturing scheme ([31]). Skew-symmetric schemes guarantee the conservation of kinetic energy in the incompressible inviscid limit. The recent KEEP-n schemes ([32]) are also available, capable of conserving (in the limit of n which tends to infinity) also the local entropy for smooth inviscid flows. Central schemes are available up to eighth order accuracy. Near the discontinuity, WENO

(weighted essentially non-oscillatory) reconstruction is used to obtain the characteristic fluxes at the cell faces which are then projected via the eigenvectors of the Euler equations. WENO schemes are available in 1st, 3rd, 5th and 7th order.

The switch between central discretization and WENO is controlled by a modified version of the Ducros shock sensor, which activates shock capture only near discontinuities.

The viscous terms are discretized with central finite difference schemes up to 8th order, or in a conservative 2nd order form where the viscous flows are evaluated first at the faces and then combined to compute the term acting at the nodes.

The time integration is based on a low-storage Runge-Kutta scheme.

STREAmS is oriented towards the simulation of canonical compressible flows under turbulent conditions, i.e. the biperiodic planar channel, the spatially evolving boundary layer and the shock-boundary layer interaction. In the present work, the tests refer to the channel, which is the simplest flow, but still involves the majority of the computationally demanding sections of the solver. In particular, we consider a calorically perfect gas and a discretization based on central KEEP-0 schemes at the 6th order, viscous schemes in Laplacian form at the 6th order and WENO schemes at the 5th order.

The reference version of the code is STREAmS-2, which is a complete rewrite of STREAmS, aiming at modularity and multibackend support. It is the latter feature that makes the OpenMP offload adaptation described in this paper relatively easy.

2.3 OpenMP porting

Since November 2015, OpenMP specifications 4.5 introduced device constructs to execute code on the "device", intended as any non-CPU device (GPU, FPGA, coprocessor) suitable for code acceleration; support for non-CPU devices has been improved and extended, with the latest OpenMP specifications available being version 5.2, released in November 2021.

OpenMP provides two different approaches to device data management: automatic data management and user-controlled data management.

Automatic data management is based on Unified Shared Memory (USM) and must be supported by the hardware. With this approach, the GPU can read and write data allocated in host memory (OpenMP terminology that refers to the CPU), and the CPU can read and write data allocated in the GPU; if the CPU and GPU do not share the same memory space, the system migrates the data, typically one page at a time, between the different memory spaces.

User-controlled data management allows the user to fine-tune data allocation and communication between the CPU and GPU, and OpenMP provides two different strategies for doing this: "data mapping" and direct allocation to GPU memory.

The data mapping strategy consists of extending the CPU allocation to the device with the OpenMP `map` clause that specifies the list of variables to be mapped. The clause can be applied to OpenMP data constructs, i.e. `target data` or `target enter data`, or to `target` compute regions. There is no need to duplicate variable declarations because host and device buffers have the same name and are "associated" in such a way that host-device synchronization is managed explicitly using the OpenMP

`target update` construct or simply by exiting the `target` compute region. Minimizing CPU-GPU memory transfers is critical to achieving good performance, and compilers may play a role in this [33].

Direct allocation relies on OpenMP device memory routines that explicitly allocate, deallocate, and synchronize data, i.e., `omp_target_alloc`, `omp_target_free`, and `omp_target_memcpy`: this approach gives the developer the most control, but unfortunately, if the same variable is needed on both the host and the device, duplicate variable declaration is required. Also, when this approach is used in Fortran code, device data must necessarily be declared as Fortran `pointer`, because the OpenMP Fortran interfaces for device memory routines (introduced in the OpenMP 5.1 specification in November 2020) only expose the `type(c_ptr)` data type (see Listing 1).

Listing 1 OpenMP Fortran interface for `omp_target_alloc` device memory routine

```
type(c_ptr) function omp_target_alloc(size, device_num) bind(c)
use, intrinsic :: iso_c_binding, only :: c_ptr, c_size_t, c_int
integer(c_size_t), value :: size
integer(c_int), value :: device_num
```

In STREAmS, we employ direct data allocation for the following reasons:

- Unified Shared Memory could impact performance
- Data duplication is not a concern because it has already been implemented for the CUDA and HIP paradigms
- Fortran pointers were already used for HIP because HIP kernels must be written in C
- Data mapping for nested Fortran-derived types is not yet well supported by modern Fortran compilers.

The Fortran interfaces for the `omp_target_alloc`, `omp_target_free` and `omp_target_memcpy` device memory routines have been wrapped into a module to provide a more Fortran-friendly interface and to avoid evaluating the size (in bytes) of the device pointer(s): the interfaces are summarized in the listings 2, 3 and 4.

Listing 2 STREAmS wrapper for omp_target_alloc

```
subroutine omp_target_alloc_f_real_4(fptr_dev, ubounds, omp_dev, ierr, lbounds)
  implicit none
  real(rkind), pointer, intent(out) :: fptr_dev(:,:,:,:)
  integer, intent(in) :: ubounds(4)
  integer, intent(in) :: omp_dev
  integer, intent(in), optional :: lbounds(4)
  integer, intent(out) :: ierr
  integer, pointer :: fptr(:)
  type(c_ptr) :: cptr_dev
  integer(ikind64) :: sizes(4), lbounds_(4)

  lbounds_ = 1
  if (present(lbounds)) lbounds_ = lbounds
  sizes = ubounds - lbounds_ + 1
  cptr_dev = omp_target_alloc(int(sizes * byte_size(1_rkind), c_size_t), int(omp_dev,
      c_int))
  if (c_associated(cptr_dev)) then
    call c_f_pointer(cptr_dev, fptr, shape=[sizes])
    fptr_dev(lbounds_(1):,lbounds_(2):,lbounds_(3):,lbounds_(4):) => fptr
    ierr = 0
  else
    fptr_dev => null()
    ierr = 1000
  endif
endsubroutine omp_target_alloc_f_real_4
```

Listing 3 STREAmS wrapper for omp_target_free

```
subroutine omp_target_free_f_real_4(fptr_dev, omp_dev)
   implicit none
   real(rkind), pointer, intent(out) :: fptr_dev(:,:,:,:)
   integer, intent(in) :: omp_dev
   type(c_ptr) :: cptr_dev

   cptr_dev = c_loc(fptr_dev)

   call omp_target_free(cptr_dev, int(omp_dev, c_int))

   nullify(fptr_dev)
 endsubroutine omp_target_free_f_real_4
```

Listing 4 STREAmS wrapper for omp_target_memcpy

```fortran
function omp_target_memcpy_f_real(fptr_dst, fptr_src, dst_off, src_off, &
   omp_dst_dev, omp_src_dev)
   implicit none
   integer(rkind) :: omp_target_memcpy_f_real
   real(rkind), target, intent(out) :: fptr_dst(..)
   real(rkind), target, intent(in) :: fptr_src(..)
   integer, intent(in) :: omp_dst_dev, omp_src_dev
   integer, intent(in) :: dst_off, src_off
   integer(ikind64) :: n_elements
   integer(c_size_t) :: total_dim, omp_dst_offset, omp_src_offset
   type(c_ptr) :: cptr_dst, cptr_src
   integer(c_int) :: omp_dst_device, omp_src_device

   n_elements = size(fptr_src,kind=ikind64)

   omp_dst_offset = int(dst_off, c_size_t)
   omp_src_offset = int(src_off, c_size_t)
   omp_dst_device = int(omp_dst_dev, c_int)
   omp_src_device = int(omp_src_dev, c_int)

   cptr_dst = c_loc(fptr_dst)
   cptr_src = c_loc(fptr_src)

   total_dim = int(n_elements * byte_size(1_rkind), c_size_t)

   omp_target_memcpy_f_int = int(omp_target_memcpy(cptr_dst, cptr_src, total_dim,
        omp_dst_offset, omp_src_offset, &
                              omp_dst_device, omp_src_device), ikind)
endfunction omp_target_memcpy_f_int
```

Listing 5 illustrates the allocation method for the array w_{GPU} used in the code to store the conservative variables: while CUDA Fortran uses the standard `allocate`, OpenMP uses the `omp_target_alloc_f` wrapper.

Listing 5 Allocation of device arrays

CUDA Fortran

```fortran
allocate(w_gpu(1-ng:nx+ng, 1-ng:ny+ng, 1-ng:nz+ng, nv))
```

OpenMP

```fortran
call omp_target_alloc_f(fptr_dev=w_gpu,ubounds=[nx+ng,ny+ng,nz+ng,nv],&
                     lbounds=[1-ng,1-ng,1-ng,1],&
                     omp_dev=self%mydev,ierr=self%ierr)
```

CPU arrays are only used during initialization and finalization, and when writing data to disk. In all of these situations, host-to-device or device-to-host data transfers are required, as seen in listing 6. For CUDA Fortran, the data can be moved using a simple Fortran assignment, while for OpenMP the custom `omp_target_memcpy_f` must

be used. The two zeros refer to the source and destination pointer offsets, while the last two dummy arguments are the destination and source device IDs, respectively.

Listing 6 CPU to GPU transfers

CUDA Fortran

```
w_gpu = w_cpu
```

OpenMP

```
self%ierr = omp_target_memcpy_f(w_gpu,w,0,0,mydev,myhost)
```

Regarding kernel implementation, it is worth noting that OpenMP does not yet have a concept of "kernel" like CUDA Fortran. A recent paper by [34] introduced kernel terminology and added preliminary kernel support to OpenMP, but according to the OpenMP specifications, any loop can be executed on the device by simply decorating it with a `target` construct that transfers control to the device.

To parallelize loop iterations (as has always been done on the CPU), they should be distributed among OpenMP teams and then executed in parallel by `num_teams` leagues of threads, as shown in listing 7. The `teams` construct creates a league of teams (the number of teams is decided by the compiler implementation), then the `distribute` construct distributes loop iteration chunks among the teams, and the `parallel do` activates multiple threads in each team to execute the iteration chunks assigned to the team. In addition, loop iterations can be "collapsed" using the `collapse` clause to expose more parallelism to the device. Loop order may follow the canonical CPU loop order (for Fortran, innermost loop on contiguous data), at least in an early stage of development; we will discuss the best OpenMP loop order in section 3.1.

Listing 7 STREAmS wrapper for omp_target_free

```
subroutine euler_x_hybrid_kernel(nv,nv_aux,...)
...
real(rkind), dimension(5,5) :: el, er
real(rkind), dimension(5) :: evmax, fi
real(rkind), dimension(5,8) :: gp,gm
...
!$omp target data map(alloc:el,er,evmax,fi,gp,gm)
!$omp target teams distribute parallel do collapse(3) has_device_addr(w_aux_gpu,fhat_gpu...)
       private(el,er,evmax,fi,gp,gm)
do k = 1,nz
  do j = 1,ny
    do i = +istart_face-1+1,iend_face
      ...
      call compute_roe_average(nx, ny, nz...)
    enddo
  enddo
enddo
!$omp end target data
endsubroutine euler_x_hybryd_kernel
```

Other OpenMP offload kernel features can be observed from the listing 7: the `has_device_addr` clause indicates that all variables listed inside already have a device address and can therefore be accessed directly from a device. Also, routines called inside a target region (as `compute_roe_average` there) must be "decorated" with the `declare target` attribute to generate both host and device code during compilation. Local data, as eigenvector matrices `el` and `er`, can be manually mapped to the device using the `target data` construct together with the `map` clause: all variables listed inside will be allocated (due to the `alloc` map type) on the device for the entire extent of the `target data` region, and no data will be transferred in or out.

In the STREAmS solver, Euler flux computations and MPI communication during boundary condition imposition can be overlapped if the asynchronous algorithm is chosen. This may be particularly useful for multinode runs where communication overhead can be reduced. In CUDA Fortran, the computation is sent in one CUDA stream and the MPI communication is sent in a separate stream to ensure overlap. In OpenMP, asynchronicity can be achieved in many ways, for example, each `target` region is a task that can be executed in parallel with some other tasks just by using the `nowait` clause. In the STREAmS solver, we chose a simpler but very effective approach using the OpenMP `sections` construct, as shown in listing 8. The `euler_x` and `bcswap` procedures are treated as two separate entities that can be executed in parallel, depending on available resources. The execution is then synchronized at the end of the `parallel sections` part, where the OpenMP runtime guarantees that all sections must be completed.

Table 2: Florence and Aurora PVC configurations

	Florence	Aurora
Vector Engines	512	448
TDP [W]	600	500
Firmware	CHECK	CHECK

Listing 8 STREAmS OpenMP asynchronous algorithm

```
call self%base_omp%bcswap(steps=[.true.,.false.,.false.])
!$omp parallel sections
!$omp section
call self%euler_x(lmax+1,nx-lmax,lmax,nx-lmax,do_update=.false.)
!$omp section
call self%base_omp%bcswap(steps=[.false.,.true.,.true.])
!$omp end parallel sections
call self%compute_aux(central=0, ghost=1)
```

2.4 PVC-powered supercomputers

For this PVC performance study, we used two different clusters: Florence and Aurora.

The first one is a small (17 nodes) internal Intel test cluster: each node, based on Lenovo SD650-I V3 platform, is equipped with dual socket Intel Sapphire Rapids 8480+ CPU with 512 GB DDR5 RAM @4800 MHz and four Intel Data Center GPU Max 1550; GPUs are connected using X^e links, while nodes are connected using Mellanox HDR 200 Gbit/s IB fabric. Software stack available is Intel oneAPI 2024.0 (compilers, oneMKL and MPI).

The first is a small (17 nodes) internal Intel test cluster: each node, based on Lenovo SD650-I V3 platform, is equipped with dual socket Intel Sapphire Rapids 8480+ CPU with 512 GB DDR5 RAM @4800 MHz and four Intel Data Center GPU Max 1550; GPUs are connected via X^e links, while nodes are connected via Mellanox HDR 200 Gbit/s IB fabric. The available software stack is Intel oneAPI 2024.0 (compilers, oneMKL and MPI).

Aurora is one of the first US exascale supercomputers, installed at the Argonne Leadership Computer Facility (ALCF), a Department of Energy Office of Science User Facility at Argonne National Laboratory. Each node, based on HPE Cray EX supercomputer platform, is equipped with dual socket Intel Sapphire Rapids 9480 with 64GB HBM2e and 512 GB DDR5 @4800 MHz each and six customized Intel Data Center GPU Max 1550; GPUs are connected using X^e links, while nodes are connected using HPE Slingshot 11 in Dragonfly topology. Software stack available is HPE Cray EX plus Intel enhancements (oneAPI 2024.0 compilers and oneMKL), together MPICH 52.2 library.

In table 2 the main characteristics of Florence and Aurora PVCs are reported: both are Intel GPU Max 1550, but Aurora PVCs have slightly less vector engines

Compiler	
Florence & Aurora	Intel Fortran compiler ifx 2024.0.0
MPI library	
Florence	Intel MPI 2021.11
Aurora	MPICH 52.2
Environment variables	
Florence & Aurora	OMP_NUM_THREADS=8 OMP_PROC_BIND=close OMP_PLACES=cores OMP_TARGET_OFFLOAD=MANDATORY LIBOMPTARGET_LEVEL_ZERO_COMPILATION_OPTIONS="-ze-opt-large-register-file"
Florence	I_MPI_OFFLOAD=1 I_MPI_OFFLOAD_TOPOLIB=none I_MPI_OFFLOAD_DOMAIN_SIZE=1 I_MPI_OFFLOAD_FAST_MEMCPY_COLL=1 I_MPI_OFFLOAD_CBWR=0 I_MPI_PIN_CELL=core I_MPI_PIN_DOMAIN=omp

Table 3: Environment configurations for Florence and Aurora PVC-powered supercomputers

active (448 vs default 512) and is set with a lower TDP (500W vs default 600W): the effects of this configuration on GPU performances will be analyzed in section 3.2.

The table 2 lists the main characteristics of the Florence and Aurora PVCs: both are Intel GPU Max 1550, but the Aurora PVC has slightly fewer vector engines active (448 vs. default 512) and is set to a lower TDP (500W vs. default 600W). The effects of this configuration on GPU performance are analyzed in the section 3.2.

In table 3 environment setup for Florence and Aurora clusters are reported: C and Fortran compilers are the same, provided by Intel oneAPI 2024.0.0, but as already discussed before, Florence uses Intel MPI library (2021.11 version) while Aurora uses Argonne MPICH library (52.2 version); the environment is therefore slightly different, and Intel MPI environment variables (`I_MPI_`*var*) are set only for Florence.

In detail, the first four environment variables are related to OpenMP: they set the number of threads, activate thread binding to CPU, specify that OpenMP thread must be placed on CPU physical cores and force the program to stop if a device construct or a device memory routine is encountered and no device is available or the available device is not supported by the implementation.

The fifth environment variable concerns Level-Zero compiler and increases the number of registers available to threads. By default, Intel compilers convert the OpenMP offload program into an intermediate language called SPIR-V and stores that in the binary produced by the compilation process. The code can be run on any hardware platform by translating the SPIR-V code into the assembly code of the platform at runtime (Just In Time - JIT compilation).

On Florence, 8 additional Intel MPI environment variables are set: the first enables handling of device buffers in MPI functions, the second prevent Intel MPI from topology definition, leaving full control to the developer; `DOMAIN_SIZE` environment variable control the number of base units per MPI rank: in this case, one MPI rank is assigned

to one device (PVC tile), `FAST_MEMCPY` enable fast memcpy functions to optimize performance for small message sizes and `CBWR` controls reproducibility of floating point operations: if set to zero, the control is disabled. Last two environment variables are related to MPI process pinning to the CPU: CPU core is selected as the minimal processor cell and the size of MPI domanin is defined (a non-overlapping subset of logical processor in a node) as the number of OpenMP threads.

2.5 OpenMP automation

The automatic generation of code supporting HPC programming paradigms is a consolidated strategy that is still used in various contexts (see [35] for a recent CFD example). A portability tool for STREAmS-2, PyconvertSTREAmS, was originally proposed in [30]. This tool, now called `sutils` (short for STREAmS utilities), is enhanced to automatically translate the CUDA Fortran backend to OpenMP. The development of `sutils` still follows its original philosophy – partly borrowed from [36] – which is to closely follow updates to the CUDA Fortran backend and provide a user-friendly backend generation of choice (CPU, HIPFort, or OpenMP). The CUDA Fortran development follows a well-established policy that allows the conversion to be streamlined and error-free. The tool can therefore generate simple and readable code for all mainstream GPU architectures along with a standard CPU implementation, with an option for OpenMP support.

However, the updated tool has some improvements over its predecessor. The main idea behind these enhancements was to provide a robust framework to continuously integrate and support new backends such as the one proposed in this paper. First, the tool has been adapted to make heavy use of the Python-based Mako template library to seamlessly set up appropriate templates for specific kernel generation in both Fortran (CPU/OpenMP) and C++ (HIPFort) programming languages. Listing 9 gives a general Mako template for generating OpenMP Offload based on information extracted from CUDA Fortran kernels. A more detailed background on this extraction process is discussed in [30].

Listing 9 Mako template for the OpenMP offload

```
% if kernel_type == "global":
  ## If local arrays exist in the global kernel, map them to the device
  % if local_arrays == True:
    !$omp target data map(alloc:${",".join(larrays)})
  % endif

  ## Final construct of directive specifying how the loop should be parallelized
  <%
  final_construct=f"!$omp target teams distribute\
  parallel do collapse({num_loop})\
  has_device_addr({','.join(gpu_arrays)})\
  {'private('+','.join(larrays)+')'\
  if local_arrays == True else ''}\
  {'&' if is_reduction == True else ''}"
  %>\

  ${final_construct}

  ## If reduction exists in the kernel, specify the directive with the reduction clause
  % if is_reduction == True:
    % for redn_id,redn in enumerate(all_reductions):
      ## Type of reduction (+/max/min/...) and their scalars
      % reduction_type = redn[0]
      % reduction_scalars = redn[1]
      ## When kernel contains more than one reduction type, add &
      % if len(all_reductions) > 1 and redn_id != len(all_reductions)-1:
        !$omp& reduction(${reduction_type}:${reduction_scalars}) &
      % else:
        !$omp& reduction(${reduction_type}:${reduction_scalars})
      % endif
    % endfor
  % endif
## If it is a device kernel, specify the device target directive
% elif kernel_type == "device":
  !$omp declare target
% endif

% if kernel_type == "global":
  ## Parallel for loops are created here in Fortran syntax
  % for idx in range(num_loop):
    do ${index_list[idx]}=${id_range[idx][0]},${id_range[idx][1]}
  % endfor
% endif

## Add global(non-parallel)/device kernel operations
${kernel_operations}

## Loop ending
% if kernel_type == "global":
  ${'enddo\n'*num_loop}
% endif

## Marks the end of a target data region for local arrays in global kernels
% if kernel_type == "global" and local_arrays == True:
  !$omp end target data
% endif
```

In addition, a simple but efficient TOML input file is used to perform basic optimizations. This includes controlling various kernel parameters such as launch bounds,

block dimension, loop index order, and number of parallel loops. This feature was added to address cases where certain kernel parameters used in the CUDA Fortran backend may not be ideal for other backends. In the context of OpenMP development, this feature was particularly useful for generating directives with different loop index orders, allowing us to easily find the optimal configuration. Listing 10 gives an example of updating the compute_residual_cuf kernel, which is a reduction kernel. This listing has three components. First, the changes are specified for a particular backend, in this case OpenMP (specified by the key, kernel_name.omp). Second, the loop order is specified by its indices from outer to inner index (key idx). Finally, the number of loops to parallelize is specified (key num_loop). An example of how this particular kernel is translated from CUDA Fortran to OpenMP is given in the listing 11. Here, the CUDA Fortran version had all loops parallelized in the order k → j → i. Now, using the TOML input specification (from the listing 10), we get the OpenMP equivalent with 2 outer loops parallelized (j and i) along with the new order, j → i → k.

Listing 10 Input TOML file for kernel optimisation

```
[compute_residual_cuf]
[compute_residual_cuf.omp]
idx = ["j","i","k"]
num_loop = 2
```

Listing 11 Tranlsation of CUDA Fortran kernel directive to OpenMP

CUDA Fortran

```
residual_rhou = 0.0
!$cuf kernel do(3) <<<*,*>>> reduce(+:residual_rhou)
do k=1,nz
 do j=1,ny
  do i=1,nx
   residual_rhou = residual_rhou + (fln_gpu(i,j,k,2)/dt)**2
  enddo
 enddo
enddo
!@cuf iercuda=cudaDeviceSynchronize()
```

OpenMP

```
residual_rhou = 0.0
!$omp target teams distribute parallel do collapse(2) has_device_addr(fln_gpu) &
!$omp& reduction(+:residual_rhou)
do j=1,ny
 do i=1,nx
  do k=1,nz
   residual_rhou = residual_rhou + (fln_gpu(i,j,k,2)/dt)**2
  enddo
 enddo
enddo
```

3 Performance results

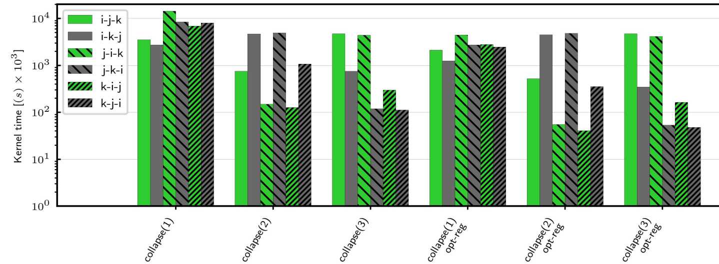
In this section, we discuss the performance obtained with the STREAmS-2 code using Intel GPU Max 1550 – hereafter simply PVC GPU – with reference to the Florence cluster cards – hereafter PVC-Florence – and to the Aurora cluster – hereafter PVC-Aurora (for details see 2.1). For the sake of comparison, in some cases we will consider the performance of NVIDIA A100 GPUs from the Leonardo cluster as well as AMD MI250X GPUs from the LUMI cluster. The analyses are performed at the different levels of parallelism at which the devices operate within their clusters. The simulated physical case is the turbulent channel flow as declared in 2.2, and the computational grids used are described in the different sections. We will refer to the offloaded computational regions as kernels, even though such terminology is not standard in OpenMP terminology.

3.1 Single tile PVC performances

Since a PVC GPU consists of two tiles, we will first look at the performance of the single PVC tile. We focus on PVC-Florence because it uses all available vector engines and has the power set to the GPU specification (600W). The simulated case uses a numerical grid of $900 \times 400 \times 500$, which is able to saturate more than 80% of the tile memory (64GB).

While OpenMP offload allows for developer-friendly GPU porting, it is still necessary to determine the optimal implementation choices in terms of achievable performance. To this end, a first study was conducted to determine the optimal order

Fig. 2: Elapsed times of convective kernel along y using central scheme considering all possible orders of loops and two levels of OpenMP loop collapse (2 or 3). PVC-Florence tile has been used.



of 3D loops among the six possible choices, e.g., i-j-k, i-k-j, j-i-k, j-k-i, k-i-j, k-j-i. Second, the three possible levels of OpenMP *collapse* were considered, i.e., 1, which corresponds to no collapsed loop, 2, and 3. Finally, the activation of the large register option (see table 3) was tested for each case. Figure 2 shows the execution times of a reference kernel – convective kernel along the y-direction and using central schemes – for all the combinations now defined.

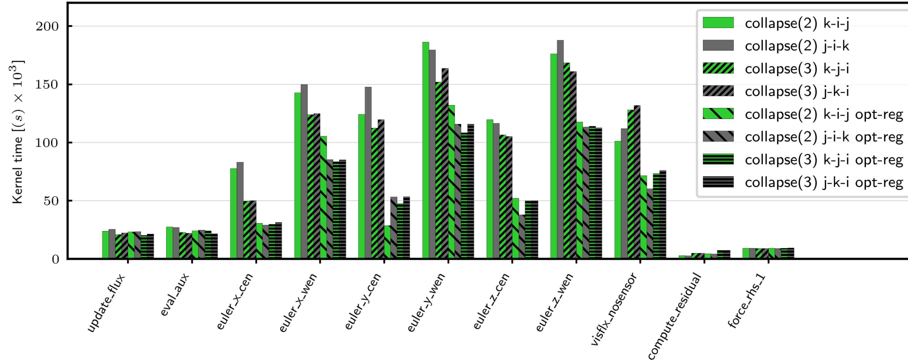
The results show that the performance obtained without collapsing the loops is dramatically worse than that of *collapse* 2 and 3, regardless of the order of the loops and the activation of large registers. In fact, it is expected that without collapsing the loops, the number of iterations of the outer loop will not be able to saturate the parallel capacity of the GPU (the number of active threads is less than the number of available vector unit threads). It should be noted that the ability to collapse loops depends on the algorithm and its implementation. STREAmS-2 has been deliberately written in a parallel-oriented way, so that for all demanding loops it is possible to collapse them completely.

Regarding the performance for *collapse* levels 2 and 3, we notice that two loop orders are significantly more efficient than the others, regardless of whether the large register option is enabled. In particular, for *collapse(2)* the optimal orderings are k-i-j and j-i-k, while for *collapse(3)* the optimal orderings are k-j-i and j-k-i. The huge performance differences between these optimal orders and the others are due to a correct/incorrect pattern of memory accesses, which turns out to be kernel-independent. In all optimal configurations, the innermost parallelized loop is the contiguous one in memory. This optimal memory access behavior replicates what can be found with other GPUs that dominate the current HPC landscape (NVIDIA or AMD).

Regarding the comparison *collapse(2)* vs. *collapse(3)*, the results are quite similar and the optimal choice is expected to be kernel dependent.

Finally, regarding the role of the option on registers, the use of large registers is definitely beneficial in all configurations. However, since register usage is a highly kernel dependent issue, there is no guarantee that what we have found for this particular kernel will hold true in general.

Fig. 3: Elapsed times of significant kernels for loop collapse 2 or 3 and, for each of them, for the two fastest loop orderings. PVC-Florence tile has been used.



In Figure 3, the previous analysis is extended to 11 significant kernels, but considering only the optimal memory access configurations, i.e. collapse(2) with k-i-j and j-i-k ordering and collapse(3) with k-j-i and j-k-i. The selected kernels represent the different types of STREAmS-2 kernels, i.e. simple (update_flux) or relatively simple (eval_aux) kernels, complex kernels that do not require the use of private arrays (visflx_nosensor), complex kernels that use private arrays (convective kernels considered in central mode, e.g, euler_x_cen, and WENO, e.g. euler_x_wen), kernels including reductions (compute_residual or force_rhs_1). The reported kernels also represent over 75% of the total computation time for the test case considered in this discussion.

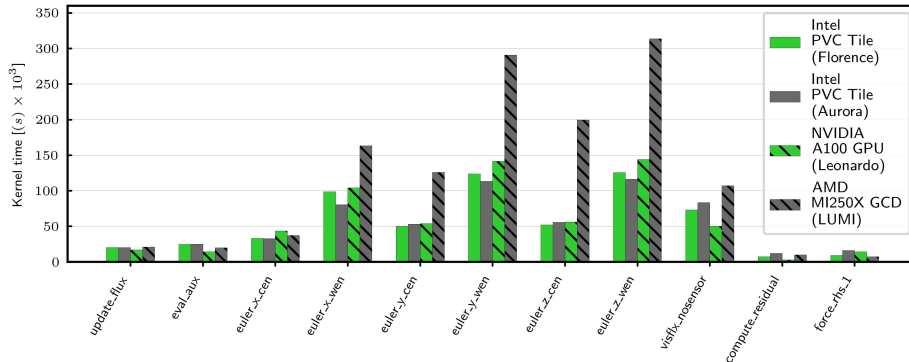
First, we observe that, as expected, enabling large registers is beneficial for complex kernels, while for simple kernels the register factor is irrelevant, or even detrimental in the case of a specific reduction kernel. However, this negative contribution is negligible in the overall balance. Since it is only possible to enable/disable large registers globally, the overall enable is definitely recommended.

Similarly, the simplest kernels show no macroscopic differences in the choice of collapse level and loop order. On the other hand, many complex kernels show clear optimal choices – e.g., collapse(2) j-i-k for euler_z_cen – or, on the contrary, clear inefficient choices – e.g., collapse(2) k-i-j for euler_x_wen. For convective kernels, there is no general superiority of collapse(2) or collapse(3), and especially in the case of collapse(2), there is no simple rule to determine the best order. For the viscous kernel, the optimal configuration is collapse(2) j-i-k, which is what was found for the CUDA Fortran kernel on NVIDIA A100 GPUs. It is worth noting that the code is mostly used in hybrid mode (the convective kernels work partly in central mode, partly in WENO mode), with typically a large majority of points evolved with the central algorithm. The optimization choice must therefore consider a compromise between the central and WENO optimal configurations, but giving more weight to the central case. The analysis of the results shows that the optimal choices of the central mode are also efficient in the WENO mode, except in the y direction, where a partial performance penalty has to be accepted in the WENO configuration. Therefore, from now on we

will consider the optimal configuration for each kernel, taking into account the optimal central mode configuration for convective kernels.

At the end of this type of optimization, it is worth noting that an a priori choice of collapse(3) k-j-i for all loops allows the user to achieve a final performance of the iteration time that is less than 10% slower than the loop-by-loop optimized version. It can therefore be a simple general recommendation for a priori loop tuning in cases where extensive testing as done thus far is not appropriate.

Fig. 4: Elapsed times of significant kernels. Comparison of different GPU devices is shown, namely PVC-Florence and PVC-Aurora files, A100 GPU from Leonardo cluster, and AMD MI250X GCD from LUMI cluster.

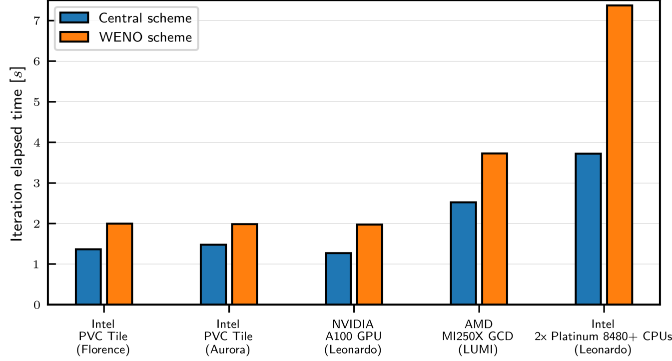


In Figure 4 we report the performance of the different kernels comparing a PVC tile from Florence, from Aurora, an A100 GPU from Leonardo and a MI250X GCD from LUMI. Overall, the performance of the PVC tiles and the A100 GPU are comparable, with a slight advantage of the PVC Florence tile for the convective kernels, while the A100 GPU prevails for the diffusive terms, possibly due to a better use of the cache. The MI250X GCD, on the other hand, shows a performance comparable to the other units for simple kernels, while for more complex kernels the gap is considerable, even exceeding a factor of two in some cases. This suggests that while the AMD GCD has theoretical performance similar to PVC tile and A100 GPU, especially in terms of bandwidth (see Table 1) which is expected to play the major role, in practice it is more difficult to achieve good performance on complex kernels.

Comparing PVC-Florence and PVC-Aurora tiles, they differ in both active vector units and power capping. When using a single tile, the lower number of active vector units of Aurora is expected to lead to a performance penalty, while power capping, which works at the GPU level (2 tiles), should not play a role. In addition, aspects related to the different software stack may slightly explain the results.

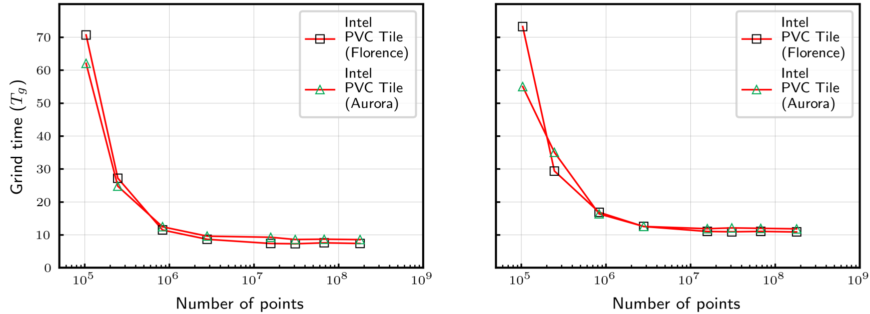
In figure 5 we report the iteration times comparing a PVC tile from Florence, from Aurora, an A100 GPU from Leonardo, and a MI250X GCD from LUMI. We also report the performance of a general purpose node (CPU) from Leonardo equipped with 2 Intel Platinum 8480+ CPUs. Consistent with what has been discussed, the

Fig. 5: Elapsed times per iteration considering full-central or full-WENO executions. Comparison of different computational units is shown, namely PVC-Florence and PVC-Aurora tiles, A100 GPU from Leonardo cluster, and AMD MI250X GCD from LUMI cluster, Leonardo General Purpose partition node equipped with 2x Intel Platinum 8480+ CPUs.



PVC tiles show similar performance to the A100 for both full-central and full-WENO runs, while the AMD GCD is about twice as slow. The CPU node based time is about 4 times slower than PVC Florence for the WENO case, while it is about 3 times slower for the central case.

Fig. 6: Elapsed times per iteration against the size of numerical grid considering full-central (left) or full-WENO (right) execution. Performance of PVC-Florence and PVC-Aurora tiles are compared.

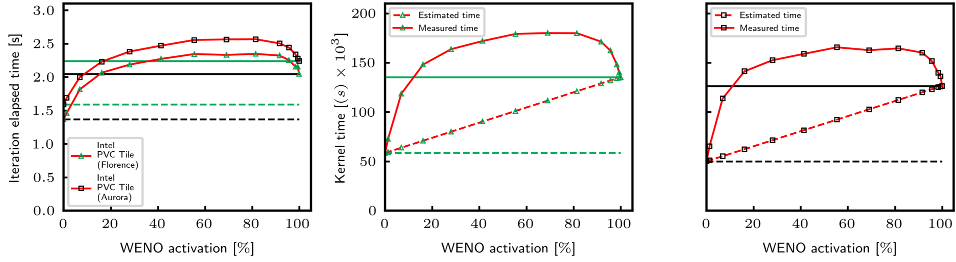


We complete the performance analysis of the single PVC tile by analyzing the effect of two simulation inputs that have a particularly significant computational impact. The first input is the number of points of the numerical discretization, which is relevant because it is known that GPUs are only efficient when they have sufficient computational load. Figure 6 shows the trend of the grind time as the number of grid

points varies. The grind time is necessary to compare computation times that refer to different grids and is defined as $T_g = T/N_{points} \times 10^6$.

The results show that for both central convective runs (Figure 6.a) and WENO runs (Figure 6.b), optimal performance is obtained when the cardinality of the grid exceeds one million points. The behavior of the Florence and Aurora tiles is very similar. The behavior is also similar to that of NVIDIA and AMD GPUs, with the difference that these cards require a number of points around 2 million to work efficiently [30].

Fig. 7: Performance of hybrid central/WENO runs. (Left) Iteration elapsed time is plotted against the percentage activation of WENO scheme. Horizontal lines represent full-central (dashed line) or full-WENO (solid line) times. PVC-Florence and PVC-Aurora tiles are compared. (Center/Right) Elapsed time of convective kernel along y using PVC-Aurora/PVC-Florence. Horizontal lines represent full-central and full-WENO times. Additional line represents the expected hybrid time resulting from weighted combination of full-central and full-WNEO times.

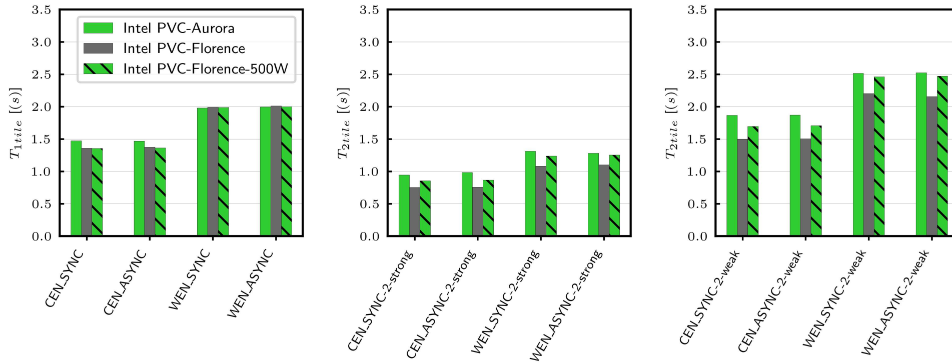


The final analysis focuses on the hybrid convective discretization, i.e. where the convective kernels operate in central or WENO mode depending on a shock sensor variable. In figure 7, on the left, the elapsed iteration times are plotted against the WENO activation percentage, for PVC-Florence and PVC-Aurora tiles. For visualization purposes, two horizontal lines are shown, corresponding to the full-central and full-WENO times, respectively. It can be seen that for WENO activations above 20%, the hybrid mode time exceeds the WENO time due to branch divergence within the kernel. However, it always remains well below the sum of the central and WENO times. The behavior of Aurora and Florence is similar, with the advantage of Florence being maintained for all activation levels. In the same figure, the elapsed times of the convective kernel along y are plotted separately for Aurora (center) and Florence (right). The line combining the central and WENO times according to the activation percentage is also shown. The cost of loop divergence is measured by the difference between the measured time and that of this straight line of interpolated values. This cost is very high, but in line with what is found with NVIDIA or AMD GPUs.

3.2 Intra-GPU scalability

As mentioned above, a PVC contains two units called tiles. A common way to use them is to associate each tile with an MPI process. In this way, from the user's point of view, each tile is used as a separate device, and the same happens from the job scheduler's point of view, which allocates the tiles as separate entities. However, since two tiles of the same GPU belong to the same physical device, it is interesting to evaluate the performance variability by switching from one to two tiles of the same PVC GPU.

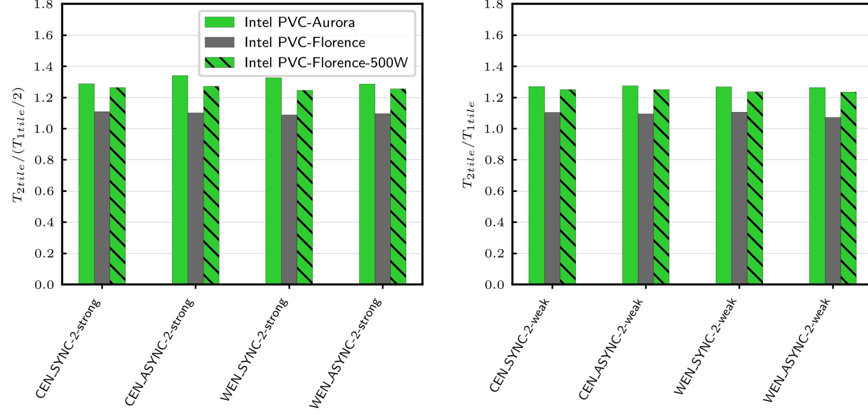
Fig. 8: Iteration elapsed times comparing 1 tile (left), 2 tiles in strong scaling spirit (center), 2 tiles in weak scaling spirit (right). Full-central/full-WENO (CEN/WEN) and synchronous/asynchronous (SYNC/ASYNC) runs are considered. Performances of PVC-Aurora, PVC-Florence, and PVC-Florence-500W are compared.



In Figure 8, the elapsed times per iteration are shown using 1 tile, using 2 tiles with the same total grid (strong scaling spirit), and using 2 tiles with the same grid per tile (weak scaling spirit). We consider runs in central mode, in WENO mode, and considering synchronous and asynchronous parallelization. For each case, the results for PVC-Aurora and PVC-Florence are compared. A third device, PVC-Florence-500W, is also considered, i.e., a Florence PVC configured at 500W to match Aurora's power limit. Considering a single tile, as seen in the previous section, PVC-Florence shows an advantage over PVC-Aurora of about 10% for central runs, while this difference practically disappears for WENO runs. With two tiles, the advantage increases and is clearly visible even for WENO runs. The results for PVC-Florence limited to 500W help to understand the previous results. At one tile, PVC-Florence-500W shows identical performance to PVC-Florence, while PVC-Aurora remains slower, presumably due to the lower number of active vector units. However, with two tiles, PVC-Florence-500W shows a significant performance degradation. To better evaluate the extent of this degradation, figure 9 shows the two-tile times divided by the expected ideal times, i.e. $T_{2tile}/(T_{1tile}/2)$ for strong scaling and T_{2tile}/T_{1tile} for weak scaling. This figure shows a good similarity of the scaling of PVC-Aurora and PVC-Florence-500W, from

which it can be deduced that the power capping plays a crucial role in the scalability limit between 1 and 2 tiles of PVC-Aurora (and PVC-Florence-500W).

Fig. 9: Results for 2 tiles reported in Figure 8 are here reported normalized by the ideal values, i.e., $T_{2tile}/(T_{1tile}/2)$ for strong scaling (left), and T_{2tile}/T_{1tile} for weak scaling (right).



As expected, the synchronous/asynchronous nature of MPI parallelization does not play a significant role at such a small number of processes. In fact, there is a slight performance penalty in the asynchronous version, presumably due to the higher complexity of the algorithm.

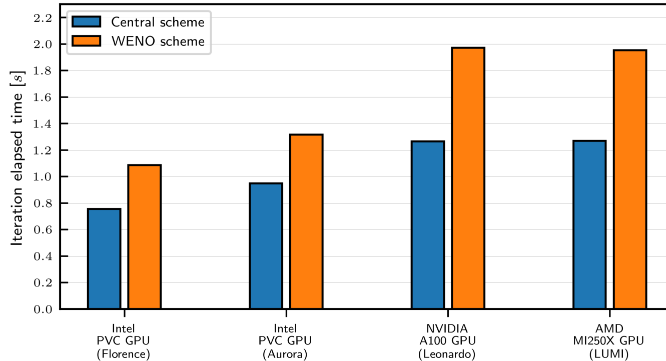
In figure 10 we present a new comparison of the elapsed times per iteration for the different GPU architectures, where the whole GPU is taken as reference, i.e. Intel PVC GPU (2 tiles), NVIDIA A100 and LUMI MI250X GPU (2 GCDs). It can be seen that the A100 and the MI250X have very similar performances in this comparison, while the PVC has significantly better performances, especially in the 600W PVC-Florence version, which has an advantage of about 40% over the A100/MI250X.

3.3 Intra-node scalability

This section analyzes the intranode scalability for Florence and Aurora nodes compared to Leonardo and LUMI GPU partitions. In line with what was discussed in the previous paragraph, the intranode scalability is measured using the entire GPU as a reference, i.e., PVC GPU, A100 GPU, and MI250X GPU. The results are shown in Figure 11 considering strong scaling (left) and weak scaling (right). To highlight the scalability, the graphs show the times rescaled with the ideal times: $T_{NGPUs}/(T_{1GPU}/N_{GPU})$ for strong scaling and T_{NGPUs}/T_{1GPU} for weak scaling. For completeness, the graph also includes the values for using 1 tile and 1 GCD, which are shown with a gray background and thus correspond to half a GPU.

In terms of weak scaling, the PVC-Florence, A100 and MI250X powered nodes show almost ideal efficiencies, while the Aurora node shows an efficiency loss of about

Fig. 10: Elapsed times per iteration considering full-central or full-WENO executions. Comparison of different GPUs is shown, namely PVC-Florence and PVC-Aurora GPUs (2 tiles), A100 GPU from Leonardo cluster, and AMD MI250X GPU (2 GCDs) from LUMI cluster.



5%. Aurora is the only architecture with 6 GPUs (corresponding to 12 tiles) per node, but the (modest) loss of efficiency does not seem to be related to this aspect, since already at 4 GPUs a non-negligible (albeit modest) performance loss can be observed. It is worth noting that the half-GPU point in this scalability is far from ideal for PVCs (especially for Aurora), while the same penalty is not found for the MI250X, which is composed of 2 GCDs.

In terms of strong scaling, the 4 architectures show essentially equivalent scalability, with efficiency losses of around 10% from 1 GPU to the full node. Overall, all architectures show more than satisfactory intra-node scalability results.

3.4 Inter-node scalability

In terms of inter-node scalability, we report the comparison between Florence and Aurora. In addition to the differences already mentioned in this section, it is useful to remember that the two clusters have a different software stack, in particular the MPI library is IntelMPI on Florence vs. mpich on Aurora. Also, the interconnect is Mellanox HDF on Florence vs HPE Slingshot on Aurora.

The performance results are shown in figure 12 for strong scaling (left) and weak scaling (right) and considering full-central and full-weno executions with synchronous and asynchronous communication modes. The reference for scalability is the computing node, but since the Florence and Aurora nodes have different numbers of GPUs, this means that an equal number of nodes does not correspond to an equal number of GPUs and MPI processes. Also, two different numerical grids are used for the single node case due to the different memory availability. Specifically, the $3200 \times 300 \times 1600$ grid is used for Florence and the $3200 \times 300 \times 2400$ grid is used for Aurora.

As with intranode scaling, the elapsed times shown are normalized to ideal times. The maximum number of nodes used in Florence (top) is 16, while in Aurora (bottom) it is 256, but for visualization purposes identical axis scales are used.

Fig. 11: Intra-node scalability, based on GPU, in the strong scaling spirit (left) and weak scaling spirit (right). Values refer to Florence (first line), Aurora (second line), Leonardo (third line) and LUMI (fourth line). Results are shown for both full-central/full-WENO (CEN/WEN) and synchronous/asynchronous (SYNC/ASYNC) runs. For Aurora, Florence and LUMI nodes, gray background region highlights half-GPU data.

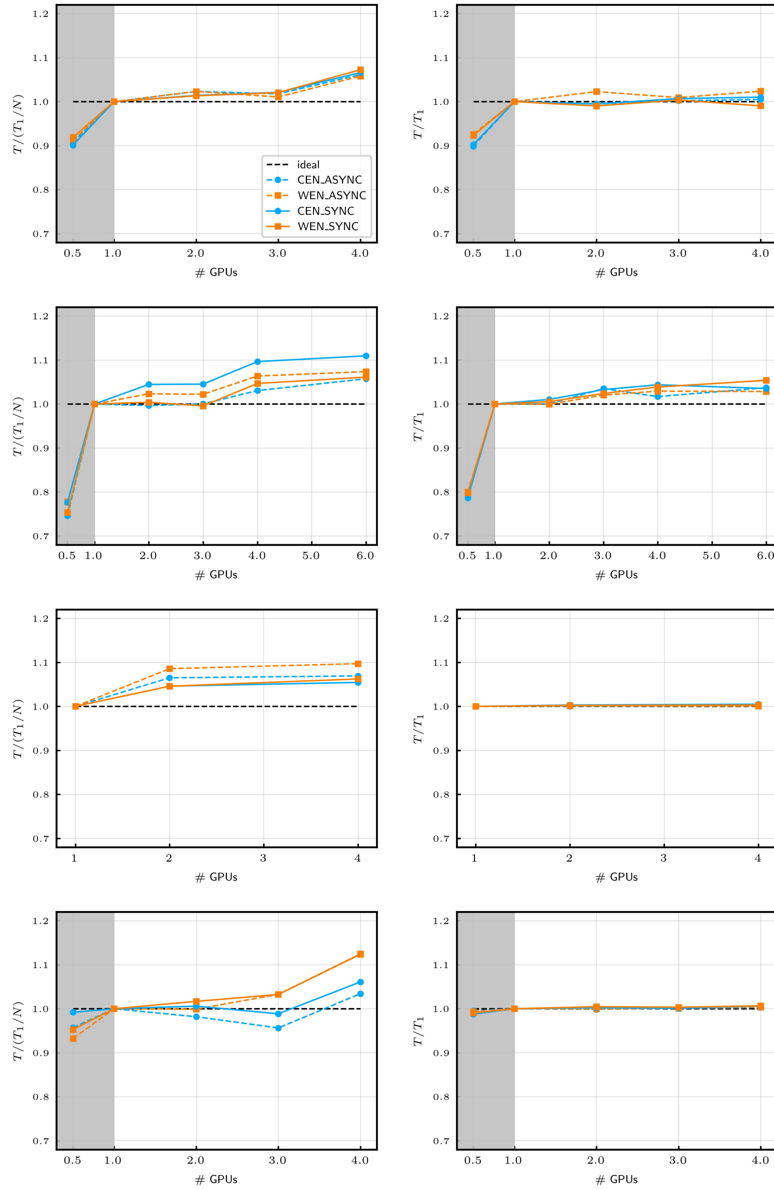
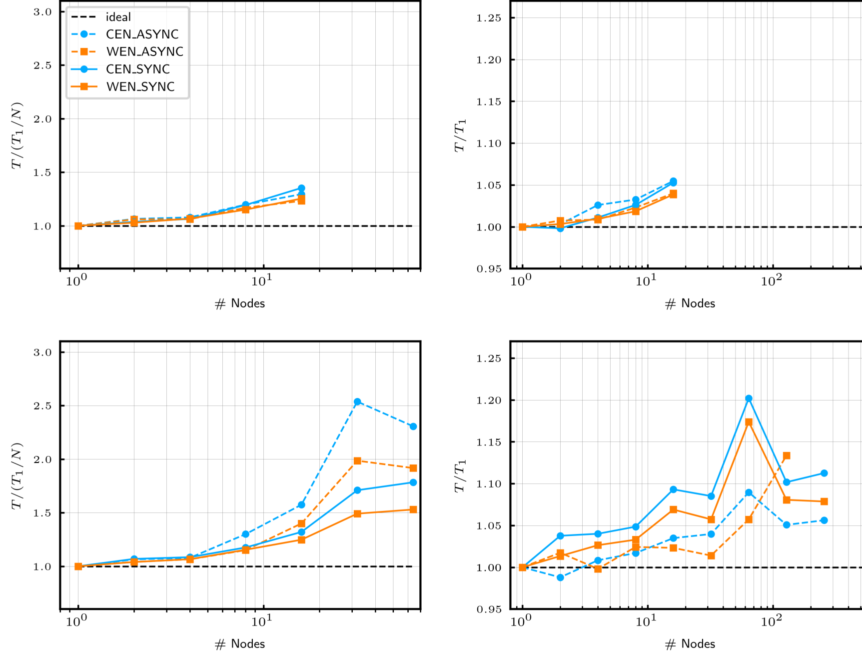


Fig. 12: Inter-node scalability based on one node for Florence (top) and Aurora (bottom) PVC-powered clusters. Strong scaling (left) and weak scaling (right) results are shown for both full-central/full-WENO (CEN/WEN) and synchronous/asynchronous (SYNC/ASYNC) runs. Despite the different maximum number of nodes, the same axis ranges are used for visualization purposes.



In the range between 1 and 16 nodes, the efficiency loss of Florence in strong scaling is about 30% for all cases, while for Aurora the loss is about 30% for synchronous cases, and around 50% for asynchronous cases. The additional complexity of the asynchronous pattern appears to be a penalty overall, despite the possibility of overlap between computation and communication. At 32 and 64 nodes, Aurora's strong scaling efficiency deteriorates significantly, as expected due to the small number of points processed by each tile. For weak scaling, up to 16 nodes, Florence shows an efficiency loss of around 5% in all cases, while for Aurora asynchrony plays an important role, halving the efficiency loss and reaching values close to 3%. Up to 256 nodes, on Aurora, the elapsed times for the asynchronous mode remain less than 10% slower than the ideal ones, while in the synchronous mode the efficiency loss approaches 20%. The different behavior, especially of the asynchronous mode, can be influenced by the different number of MPI processes, the different connection topology and also by the different MPI libraries used.

Overall, the inter-node performance of both clusters is satisfactory, with a slight advantage for Florence, partly due to the lower number of GPUs used for the same

number of nodes, which however has a much smaller overall size. The role of asynchronicity is basically detrimental in strong scaling. It is beneficial in weak scaling, but only for Aurora, which is the most important case for large-scale production contexts. However, it should be noted that in this paragraph we intentionally only reported scalability, while the real benefit of asynchronism has to be weighed against the performance penalty at a single GPU/node. All in all, asynchronism does not seem to be a real advantage in clusters like Florence or Aurora, mainly due to their optimal networking capabilities.

In terms of production simulations, we have demonstrated that Aurora is an exascale machine capable of significantly pushing the boundaries of compressible fluid dynamics, both in terms of hero runs and parametric studies of medium-sized runs. Furthermore, the ability to achieve good performance using a lightweight paradigm such as OpenMP is encouraging for the development and maintenance of current and future codes.

4 Conclusions

The Intel Ponte Vecchio (PVC) GPU presents itself as an ambitious contender in the HPC landscape currently dominated by GPU computing. In this paper, we presented challenges and results of its use considering the STREAmS community code. STREAmS is a fluid dynamics solver for compressible and turbulent canonical flows, an area traditionally greedy for computational resources to narrow the gap between direct numerical simulations (i.e., without turbulence models) and realistic contexts.

The first issue addressed was the porting of code required to use PVC. In this context, it is well known that programming NVIDIA and AMD GPUs to achieve optimal performance is traditionally challenging and may require particularly invasive interventions in the existing software. In addition, for Fortran codes such as STREAmS, porting may require additional efforts when using some paradigms, e.g., HIP for AMD does not provide a complete version dedicated to Fortran. However, compiler support for directive-based paradigms is growing and should lead to less porting effort in the future. Intel decided to invest, directly at launch of X^e architecture, in support for OpenMP, a widely used standard paradigm that in recent years has gained specific support for accelerated architectures in recent years (OpenMP v4.5 and v5.x). We discussed the OpenMP porting done on STREAmS and its implementation. The completed port is not fully based on the directive approach, but rather on a hybrid approach thanks to OpenMP memory allocation/free/copy APIs. This choice is due to the software architecture of STREAmS-2 and the use of an automatic conversion tool between backends, which was successfully extended to convert backend-dependent code parts to the new OpenMP offload backend. The realized code is highly readable and potentially usable in the future to take advantage of additional hardware devices.

The second aspect addressed was the evaluation of PVC performance using STREAmS-2 and considering the supersonic turbulent channel as the physical reference case. The performance was evaluated on two major clusters powered by PVC GPUs: the Florence cluster – a test-oriented cluster with standard PVC configuration – and the Aurora cluster – runner-up of the Top500 list and equipped with PVC

in a slightly capped configuration. The performance analyses followed the different levels of cluster parallelism: intrinsic PVC tile parallelism, comparison of single-tile PVC vs. 2-tile PVC performance within the same PVC GPU, intra-node scalability of PVC GPUs, and inter-node scalability based on compute node. The single-tile analysis revealed that while the OpenMP implementation is quite simple, it is critical to follow certain rules to achieve adequate performance. Improved performance can be achieved by tuning kernels one by one, but the benefits of this type of optimization are very limited. Overall, the PVC GPU shows about 40% better performance than the NVIDIA A100 GPU or the AMD MI250X, which however were released about 3 years earlier. In addition, the scalability shows more than satisfactory results in both strong and weak scaling spirits. Overall, the entry of PVC into the GPU computing HPC landscape is a positive step forward for the diversification and competitiveness of the industry.

Acknowledgments. Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (EuroHPC JU) and Germany, Italy, Slovenia, Spain, Sweden, and France under grant agreement No 101092621.

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357: this work was done on a pre-production supercomputer with early versions of the Aurora software development kit.

References

[1] TOP500. https://www.top500.org/lists/top500/2023/11/. Accessed: 05 March 2024 (2023)

[2] EUROHPC JU. https://eurohpc-ju.europa.eu/about/our-supercomputers_en. Accessed: 05 March 2024 (2024)

[3] LUMI. https://lumi-supercomputer.eu/. Accessed: 05 March 2024 (2024)

[4] LEONARDO. https://leonardo-supercomputer.cineca.eu/. Accessed: 05 March 2024 (2024)

[5] CUDA, 2023. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 25 February 2023 (2024)

[6] Zhu, X., Phillips, E., Spandan, V., Donners, J., Ruetsch, G., Romero, J., Ostilla-Mónico, R., Yang, Y., Lohse, D., Verzicco, R., Fatica, M., Stevens, R.J.A.M.: Afid-gpu: A versatile navier–stokes solver for wall-bounded turbulent flows on gpu clusters. Computer Physics Communications **229**, 199–210 (2018) https://doi.org/10.1016/j.cpc.2018.03.026

[7] Wei, J., Jiang, J., Liu, H., Zhang, F., Lin, P., Wang, P., Yu, Y., Chi, X., Zhao, L., Ding, M., Li, Y., Yu, Z., Zheng, W., Wang, Y.: Licom3-cuda: a gpu version of lasg/iap climate system ocean model version 3 based on cuda. The Journal of Supercomputing **79**(9), 9604–9634 (2023) https://doi.org/10.1007/s11227-022-05020-2

[8] kokkos. https://github.com/kokkos/kokkos. Accessed: 05 March 2024 (2024)

[9] RAJA. https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms. Accessed: 05 March 2024 (2024)

[10] alpaka. https://github.com/alpaka-group/alpaka. Accessed: 05 March 2024 (2024)

[11] OpenMP, 2024. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf/. Accessed: 25 February 2024 (2024)

[12] OpenACC, 2024. https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/. Accessed: 25 February 2024 (2024)

[13] SYCL 2020 Specification (revision 8). https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf. Accessed: 24 March 2024 (2024)

[14] ISO/IEC: Programming languages — technical specification for c++ extensions for parallelism. Technical report (2015)

[15] Costanzo, M., Rucci, E., García-Sanchez, C., Naiouf, M., Prieto-Matías, M.: Assessing opportunities of sycl for biological sequence alignment on gpu-based systems. The Journal of Supercomputing (2024) https://doi.org/10.1007/s11227-024-05907-2

[16] Malenza, G., Cesare, V., Aldinucci, M., Becciani, U., Vecchiato, A.: Toward hpc application portability via c++ pstl: the gaia avu-gsr code assessment. The Journal of Supercomputing (2024) https://doi.org/10.1007/s11227-024-06011-1

[17] Costa, P., Phillips, E., Brandt, L., Fatica, M.: Gpu acceleration of cans for massively-parallel direct numerical simulations of canonical fluid flows. Computers & Mathematics with Applications **81**, 502–511 (2021) https://doi.org/10.1016/j.camwa.2020.01.002

[18] HIP: C++ Heterogeneous-Compute Interface for Portability, 2023. https://github.com/ROCm-Developer-Tools/HIP/. Accessed: 25 February 2023 (2024)

[19] Jansson, N., Karp, M., Podobas, A., Markidis, S., Schlatter, P.: Neko: A Modern, Portable, and Scalable Framework for High-Fidelity Computational Fluid Dynamics. arXiv preprint arXiv:2107.01243 (2021)

[20] Germaschewski, K., Allen, B., Dannert, T., Hrywniak, M., Donaghy, J., Merlo,

G., Ethier, S., D'Azevedo, E., Jenko, F., Bhattacharjee, A.: Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU. Physics of Plasmas **28**(6), 062501 (2021)

[21] Carnimeo, I., Affinito, F., Baroni, S., Baseggio, O., Bellentani, L., Bertossa, R., Delugas, P.D., Ruffino, F.F., Orlandini, S., Spiga, F., Giannozzi, P.: Quantum espresso: One further step toward the exascale. Journal of Chemical Theory and Computation **19**(20), 6992–7006 (2023)

[22] Gavini, V., Baroni, S., Blum, V., Bowler, D.R., Buccheri, A., Chelikowsky, J.R., Das, S., Dawson, W., Delugas, P., Dogan, M., *et al.*: Roadmap on electronic structure codes in the exascale era. Modelling and Simulation in Materials Science and Engineering **31**(6), 063301 (2023)

[23] Zubair, M., Walden, A., Nastac, G., Nielsen, E., Bauinger, C., Zhu, X.: Optimization of ported cfd kernels on intel data center gpu max 1550 using oneapi esimd. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 1705–1712. Association for Computing Machinery, New York, NY, USA (2023)

[24] Owen, H., Lehmkuhl, O., D'Ambra, P., Durastante, F., Filippone, S.: Alya toward exascale: algorithmic scalability using psctoolkit. The Journal of Supercomputing (2024) https://doi.org/10.1007/s11227-024-05989-y

[25] Bernardini, M., Modesti, D., Salvadore, F., Pirozzoli, S.: STREAmS: a high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows. Comput. Phys. Commun. **263**, 107906 (2021)

[26] Bernardini, M., Modesti, D., Salvadore, F., Sathyanarayana, S., Della Posta, G., Pirozzoli, S.: STREAmS-2.0: Supersonic turbulent accelerated Navier-Stokes solver version 2.0. Comput. Phys. Commun., 108644 (2023)

[27] Modesti, D., Sathyanarayana, S., Salvadore, F., Bernardini, M.: Direct numerical simulation of supersonic turbulent flows over rough surfaces. J. Fluid Mech. **942**, 44 (2022)

[28] Bernardini, M., Della Posta, G., Salvadore, F., Martelli, E.: Unsteadiness characterisation of shock wave/turbulent boundary-layer interaction at moderate Reynolds number. J. Fluid Mech. **954**, 43 (2023)

[29] Salvadore, F., Memmolo, A., Modesti, D., Della Posta, G., Bernardini, M.: Direct numerical simulation of a microramp in a high-reynolds number supersonic turbulent boundary layer. Physical Review Fluids **8**(11), 110508 (2023)

[30] Sathyanarayana, S., Bernardini, M., Modesti, D., Pirozzoli, S., Salvadore, F.: High-speed turbulent flows towards the exascale: STREAmS-2 porting and performance. Preprint at https://arxiv.org/abs/2304.05494 (2023)

[31] Pirozzoli, S., Bernardini, M., Grasso, F.: Direct numerical simulation of transonic shock/boundary layer interaction under conditions of incipient separation. Journal of Fluid Mechanics **657**, 361–393 (2010) https://doi.org/10.1017/S0022112010001710

[32] Tamaki, Y., Kuya, Y., Kawai, S.: Comprehensive analysis of entropy conservation property of non-dissipative schemes for compressible flows: Keep scheme redefined. Journal of Computational Physics **468**, 111494 (2022) https://doi.org/10.1016/j.jcp.2022.111494

[33] Guo, H., Zhang, L., Zhang, Y., Li, J., Xu, X., Liu, L., Cai, K., Wu, D., Yang, S., Kong, L., Gao, X.: Openmp offloading data transfer optimization for dcus. The Journal of Supercomputing **80**(2), 2381–2402 (2023) https://doi.org/10.1007/s11227-023-05422-w

[34] Tian, S., Scogland, T., Chapman, B., Doerfert, J.: Openmp kernel language extensions for performance portable gpu codes. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23, pp. 876–883. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624164 . https://doi.org/10.1145/3624062.3624164

[35] Huang, K., Che, Y., Xu, C., Dai, Z., Zhang, J.: Improving cuda performance of an unstructured high-order cfd application under op2 framework. The Journal of Supercomputing **80**(5), 5832–5846 (2023) https://doi.org/10.1007/s11227-023-05679-1

[36] GPUFORT, 2021. https://github.com/ROCmSoftwarePlatform/gpufort/. Accessed: 25 February 2023 (2021)