TUDelft

Delft University of Technology

High-speed turbulent flows towards the exascale

STREAmS-2 porting and performance

Sathyanarayana, Srikanth; Bernardini, Matteo; Modesti, Davide; Pirozzoli, Sergio; Salvadore, Francesco

Important note
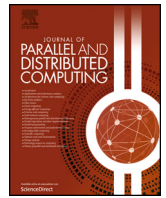To cite this publication, please use the final published version (if applicable).
Please check the document version above.

High-speed turbulent flows towards the exascale: STREAmS-2 porting and performance

Srikanth Sathyanarayana [a,e], Matteo Bernardini [a], Davide Modesti [b,d,*], Sergio Pirozzoli [a], Francesco Salvadore [c]

[a] *Department of Mechanical and Aerospace Engineering, Sapienza University of Rome, via Eudossiana 18, 00184 Rome, Italy*
[b] *Aerodynamics Group, Faculty of Aerospace Engineering, Delft University of Technology, Kluyverweg 2, 2629 HS Delft, the Netherlands*
[c] *HPC Department, Cineca, Rome office, via dei Tizii 6/B, Rome, Italy*
[d] *Gran Sasso Science Institute, Viale Francesco Crispi 7, 67100 L'Aquila, Italy*
[e] *Max Planck Computing and Data Facility, Gießenbachstraße 2, 85748 Garching, Germany*

ABSTRACT

Exascale High Performance Computing (HPC) represents a tremendous opportunity to push the boundaries of Computational Fluid Dynamics (CFD), but despite the consolidated trend towards the use of Graphics Processing Units (GPUs), programmability is still an issue. STREAmS-2 (Bernardini et al. Comput. Phys. Commun. 285 (2023) 108644) is a compressible solver for canonical wall-bounded turbulent flows capable of harvesting the potential of NVIDIA GPUs. Here we extend the already available CUDA Fortran backend with a novel HIP backend targeting AMD GPU architectures. The main implementation strategies are discussed along with a novel Python tool that can generate the HIP and CPU code versions allowing developers to focus their attention only on the CUDA Fortran backend. Single GPU performance is analyzed focusing on NVIDIA A100 and AMD MI250x cards which are currently at the core of several HPC clusters. The gap between peak GPU performance and STREAmS-2 performance is found to be generally smaller for NVIDIA cards. Roofline analysis allows tracing this behavior to unexpectedly different computational intensities of the same kernel using the two cards. Additional single-GPU comparisons are performed to assess the impact of grid size, number of parallelized loops, thread masking and thread divergence. Parallel performance is measured on the two largest EuroHPC pre-exascale systems, LUMI (AMD GPUs) and Leonardo (NVIDIA GPUs). Strong scalability reveals more than 80% efficiency up to 16 nodes for Leonardo and up to 32 for LUMI. Weak scalability shows an impressive efficiency of over 95% up to the maximum number of nodes tested (256 for LUMI and 512 for Leonardo). This analysis shows that STREAmS-2 is the perfect candidate to fully exploit the power of current pre-exascale HPC systems in Europe, allowing users to simulate flows with over a trillion mesh points, thus reducing the gap between the Reynolds numbers achievable in high-fidelity simulations and those of real engineering applications.

1. Introduction

High-performance computing (HPC) is of paramount importance for scientists and engineers, having become a key tool for driving advancements across various disciplines in recent years. Supercomputing, in particular, has revolutionized the field of computational fluid dynamics (CFD), enabling high-fidelity simulations of turbulence that are crucial for understanding the behavior of complex flows in both scientific and industrial applications. The capability to perform detailed simulations at unprecedented scales has been greatly enhanced by the advent of modern heterogeneous architectures, leveraging multi-node systems equipped with high-performance accelerators. Graphical Processing Units (GPUs) have been at the forefront of these changes and are the driving architecture on the current path to exascale computing.

Recent computer architectures have become increasingly heterogeneous with the use of multi-node systems equipped with high-performance accelerators. Graphical Processing Units (GPUs) have been at the forefront of these changes and are the driving architecture on the current path to exascale computing. Currently, seven of the top ten supercomputers use GPUs for acceleration [1]. Much of the success has been attributed to optimized architectures that deliver high performance per watt. However, programmability is generally a major drawback, but programming efforts over the past few years, recent standardized paradigms and performance libraries have alleviated this challenge. To this end, NVIDIA is the most widely used GPU with its popular CUDA programming framework [2]. In recent years, several legacy computational fluid dynamics (CFD) codes have been ported to NVIDIA architectures and new ones have been developed to target this backend. AFiD [3] and CANS [4] are popular open source incompressible Direct Numerical Simulations (DNS) codes. AFiD uses a CUDA Fortran [5] approach, while CaNS mostly uses OpenACC [6]. For compressible flows, PyFR [7] and ZEFR [8] are unstructured based flow solvers using CUDA for GPU acceleration. More recently, URANOS [9] has been developed using OpenACC directives and is capable of running in DNS, Large Eddy Simulations (LES) and Wall-Modelled LES (WMLES) modes. Nek5000, a popular open source pseudo-spectral code also has an OpenACC implementation [10].

An important development in recent years has been the emergence of new GPU vendors with different hardware and programming environments. This has created serious problems for developers, forcing them to maintain multiple versions of the same code targeting different GPU architectures. This is also reflected in the current exascale/pre-exascale supercomputers in the US and EU. Frontier at Oak Ridge National Laboratory, El Capitan at LLNL and LUMI at CSC are equipped with AMD GPUs, while Aurora at Argonne National Laboratory has Intel GPUs and Leonardo at Cineca has NVIDIA GPUs. This has prompted discourse on the potential of programming solutions to address the latest GPUs, regardless of their architectural specifications. It is therefore desirable that such programming solutions should be performance-portable in the strict sense, i.e., the same source code should be able to be compiled on different hardware. Alternatively, they should be implementation-adaptable, i.e., codes should provide a framework for different implementations for different hardware with minimal code duplication. Neko, part of the Nek5000 family, a spectral element based CFD code, has recently released its multi-backend version [11]. They use multi-level abstractions using abstract Fortran types, which facilitates the implementation of hardware-specific backends. PyFR uses Python-based Mako templating engine to develop their in-house domain-specific language tool to generate backends for different software paradigms. Both Neko and PyFR have the ability to target vendor specific software such as CUDA or HIP [12] and general purpose frameworks such as OpenCL [13]. OpenSBLI, a framework to study shock boundary layer interactions utilizes the OPS library to automatically produce various backends targeting both CPU and GPU architectures [14]. The unstructured flow solver charLES also has been developed to run on both NVIDIA and AMD GPUs [15].

Over the years, our compressible flow solver STREAmS has responded to the changing nature of High Performance Computing (HPC) and recognized the potential of GPUs. An earlier version of STREAmS was ported to a single GPU to target the NVIDIA Fermi architecture [16]. This was followed by a full port to CUDA Fortran (hereafter called STREAmS-1) [17]. More recently, recognizing the need to extend STREAmS-1 to support wider set of HPC architectures, a complete refactoring called STREAmS-2 was developed to support multi-backend and multi-equation applications in an object-oriented manner [18]. This continuous evolution of the solver is mainly due to its maturity resulting from a history of more than 20 years dedicated to the study of compressible wall-bounded flows. Several seminal DNS studies on wall-bounded canonical flows such as supersonic and hypersonic boundary layers [19][20][21], shock boundary layer interactions [22], supersonic internal flows [23] have been performed. In addition, the solver has been extended to study some of the more challenging problems such as the supersonic roughness-induced transition [24][25] and the effects of distributed surface roughness [26].

In this work we extend the capabilities of STREAmS-2 by developing a HIP backend to support AMD GPUs in addition to the already available NVIDIA and CPU backends. We begin with a brief background on the governing equations and numerical methods that form the basis of the algorithms used in the solver. We then provide a detailed description of the porting strategies used to develop the HIP variant of STREAmS-2. Finally, we present results based on single-GPU performance on different architectures, followed by scalability results based on multi-node, multi-GPU HPC clusters.

2. Numerical methods

STREAmS-2 solves the compressible Navier-Stokes equations for an ideal gas in Cartesian coordinates using a finite difference discretization. The nonlinear terms are treated using a hybrid discretization that switches between a central scheme in smooth flow regions and a shock-capturing scheme in shocked regions. Numerical stability in smooth flow regions is achieved by using the built-in anti-aliasing properties of the skew-symmetric form of the convective terms, which are cast in terms of numerical fluxes to allow easy hybridization with the shock capturing scheme [27]. This formulation guarantees discrete conservation of kinetic energy in the inviscid incompressible limit. Optionally, one can also choose the latest KEEP-n scheme [28], which also guarantees local entropy conservation for inviscid smooth flows.

In the vicinity of discontinuities, a weighted essentially non-oscillatory (WENO) reconstruction is used to obtain the characteristic fluxes at the cell faces, which are then projected onto the right eigenvectors of the Euler equations. The switch between the two discretizations is controlled by a modified version of the classical Ducros shock sensor, which activates the shock capturing algorithm only close to discontinuities [17]. The implementation is general enough to allow an arbitrary order of accuracy, although at the moment discretizations up to 8th and 7th order are available for the central scheme and WENO reconstruction respectively. Viscous terms are expanded in Laplacian form and discretized with central finite difference formulae up to an eighth order of accuracy. For time integration we use a third-order low storage Runge-Kutta scheme.

A feature of STREAmS-2 is that both calorically and thermally perfect gases can be simulated, whereby the dependency of the specific heats on temperature is accounted for using NASA polynomials [29].

Apart from this last feature, the numerical approach described here has been used by the group for two decades to study compressible wall-bounded flows. The main novelty of STREAmS-2 is that the legacy solver has been rewritten using an object-oriented flavor to allow future extensions and porting to different backends, as described in the next section.

3. GPU porting

The prevalence of GPUs in HPC has increased exponentially over the past decades, largely due to their capacity to deliver substantial performance. With the emergence of new vendors, developers have greater flexibility to select the direction of their code development. This could entail selecting a specific GPU hardware and programming approach that is more aligned with their numerical implementations. However, in the context of the latest pre-exascale/exascale supercomputers, which tend to use different GPU hardware, this approach is severely limited. Ideally, scientific codes should have an implementation that is capable of handling runs on any hardware. This approach will enable codes targeting GPUs to approach and

exceed the exascale limit on any architecture, thereby solving computationally expensive problems that were previously thought to be infeasible. This section outlines the methodology employed to exploit the features of STREAmS-2 in order to develop a HIP backend that allows us to target AMD GPUs. The proposed development will assist us in achieving our objective of extending the STREAmS platform to encompass the ability to handle more HPC hardware.

3.1. Code architecture

STREAmS was previously adapted to the changing HPC environment when it was ported to STREAmS-1 using CUDA Fortran to target NVIDIA GPUs, and a detailed description of the porting strategy and performance is available in the reference publication [17]. STREAmS-1 was developed in Fortran primarily for its simplicity and improved code readability. In addition, there was minimal use of external libraries and heavy reliance on standardized frameworks (e.g., MPI). This allowed us to develop the code in a vendor-independent way, using multiple compilers and running on different supercomputers. However, the code remained largely procedural, preventing easy extension to other programming paradigms. In addition, STREAmS-1 made extensive use of compiler preprocessor directives to separate the CPU and GPU (CUDA Fortran) versions of the code. This strategy is similar to other popular CFD codes [3][4], although STREAmS-1 uses a more distinct pattern to improve readability. Using the preprocessor directive approach forces developers to maintain both execution branches, which limits scaling to other architectures and seriously compromises code readability and maintainability. With the advent of newer GPU architectures, this development model needed to be adapted.

STREAmS-1 was further developed into STREAmS-2 to provide a more scalable version, which further helped to solve the porting problem [18]. The design of STREAmS-2 is based on the object-oriented Fortran 2008 standard, which allows developers to accommodate completely different implementations depending on the backend, without having to duplicate or deeply restructure the entire code. Some of the ideas in this approach are similar to those used in Neko [11], although a more fine-grained object-oriented design is used in that case.

STREAmS-2 has been developed using CUDA Fortran to target NVIDIA GPUs, however, to extend the solver to other backends, five approaches were considered:

1. Vendor specific paradigms: CUDA/HIP/OneAPI [30]
2. OpenCL standard
3. Directive-based standards: OpenMP [31]/OpenACC
4. Other high-level porting approaches: Kokkos [32]/Legion [33]/AdaptiveCpp [34]/alpaka [35]/RAJA [36]
5. Intrinsic language constructs (*do concurrent* from Fortran or *parallel stl* from C++)

Each of these strategies has its advantages and disadvantages. Option 3 is attractive because it is standardized, multi-device capable and high level, but the implementation is still lagging behind and advanced optimizations of complex kernels can be difficult. The same goes for 5, where the implementation is currently even more inadequate. Approach 2 is potentially good, but the resulting code is quite verbose and optimal performance is difficult to achieve. Option 4 is promising, but requires significant code rewrites, and support for different devices can be a critical issue.

In general, vendor-specific paradigms are still the best choice for achieving optimal performance. Since our specific goal is to enable the code to run on a large subset of modern HPC centers, the main targets are CUDA (NVIDIA) and HIP (AMD). The CUDA and HIP paradigms are essentially very similar, and the hipify [37] tool makes it easy to convert code from CUDA to HIP. However, there are some critical issues with a Fortran based implementation. CUDA has a Fortran specific variant, CUDA Fortran, while HIP only offers HIPFort [38], which is basically a collection of Fortran variables and interfaces to HIP library functions. This means that at least two main strategies can be adopted:

1. Develop using CUDA Fortran and periodically translate the code to the HIP paradigm.
2. Develop a C/C++ implementation of the CUDA parts and write Fortran interfaces to call them. This means that hipify can be used to convert CUDA code to HIP.

Strategy 1 makes CUDA Fortran development much easier, but HIP translation is difficult. Strategy 2 makes CUDA development harder, but HIP compilation easier. Given the normal development cycle of the code, which includes implementation of new methods and physics, we believe that an easier development approach is crucial. CUDA Fortran seems to be the best compromise between code readability and resulting performance. Therefore we decided to follow strategy 1. Since STREAmS-2 already uses an implementation of CUDA Fortran for the NVIDIA backend, this strategy essentially means porting the existing kernels directly to HIP to target the AMD backend. Note that the HIP backend can in principle also target the NVIDIA backend, but in this work we focus our development mainly on handling the AMD GPUs.

So far we have not mentioned the traditional CPU backend, which is generally important to support in a scientific code. We use a similar strategy for developing the CPU backend using the CUDA Fortran version, although the approach is much simpler due to the many similarities between them.

The final structure of the updated STREAmS-2 including support for three backends (CPU, CUDA Fortran and HIP) is shown in Fig. 1. All the backends begin with a main program, which is essentially the entry point for running a particular backend and governing equation system. Further program structure is based on the following considerations,

Equation and backend independent code

This consists of the Field and Grid objects. The Grid object primarily uses procedures for general grid management, which mainly correspond to I/O routines and evaluation of grid metrics. The Field object, on the other hand, represents a generic field development of the solver along with I/O management. This part of the code also consists of a parameter module, which is mainly used to define precision, constants and other utility functions.

Equation Independent and backend dependent code

This mainly concerns generic MPI communication procedures implemented according to a particular backend. The Base ⟨backend⟩ blocks in Fig. 1 represent this. Both the CUDA Fortran and HIP backends have some form of GPU-aware transfers, while this aspect is irrelevant in the CPU backend.
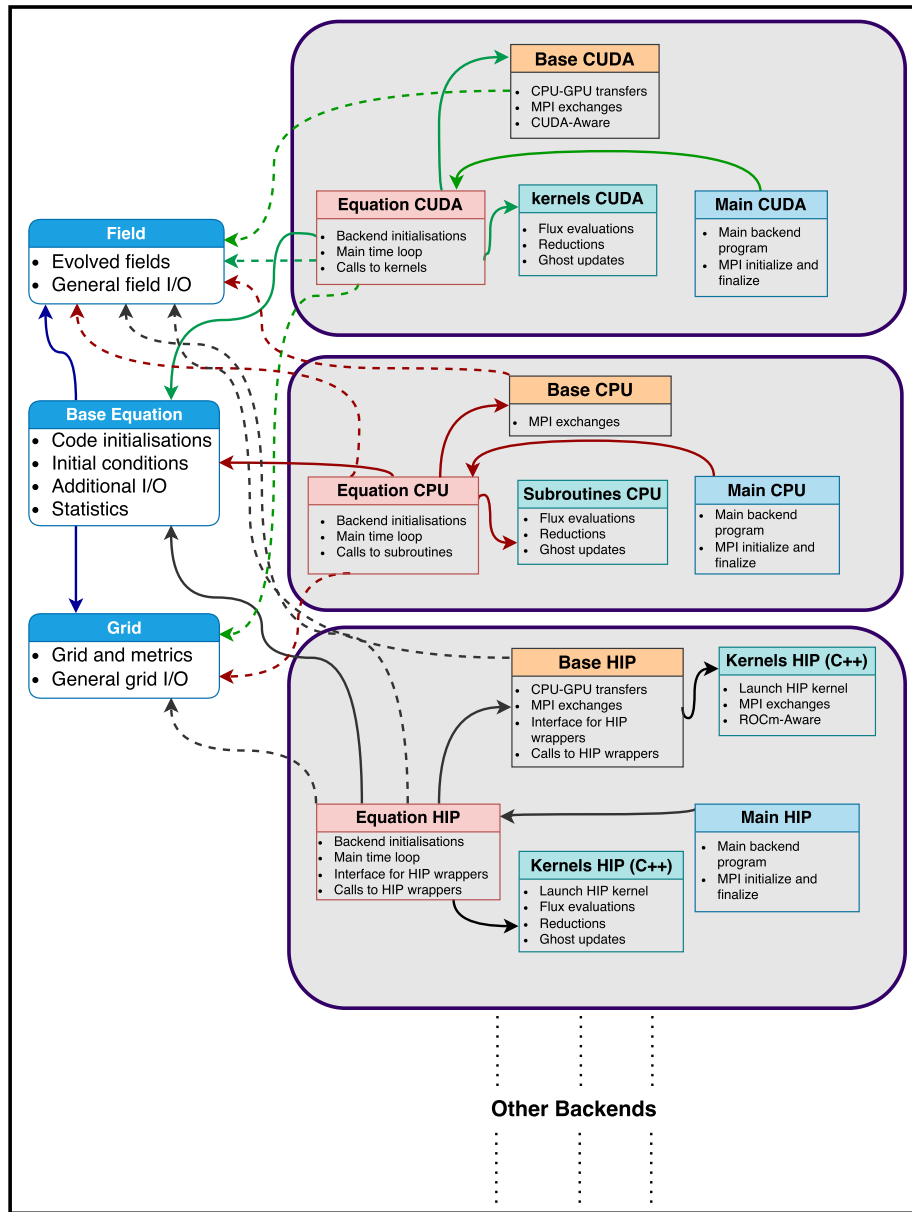
Fig. 1. Program flow of STREAmS-2 with CUDA Fortran, HIP and CPU backends. This is an example of a single equation type with multiple program backends. Blocks with labels represent Fortran objects or modules: Field and Grid (equation and backend independent), Base Equation (equation dependent and backend independent), Base CUDA/CPU/HIP (equation independent and backend dependent) and other blocks that are both equation and backend dependent. On the right, the gray container boxes group the blocks used by each backend. Each line represents an object that uses another object. Solid lines represent objects that contain other objects. Dashed lines represent objects that use other external objects but are saved internally as pointers.

Equation dependent and backend independent code

Next, the Base Equation block in Fig. 1 is dedicated to the initialization of the flow and to perform all typical backend independent operations (e.g., statistics preparation, transport properties definition) for a particular equation type. This block can generally be extended to systems of other governing equations.

Equation and backend dependent code

The Equation ⟨backend⟩ block contains the main timeloop for a particular equation type. This block also varies for different backends. For example, the Equation CUDA block consists of all calls to kernels written in CUDA Fortran. For the CPU backend, these kernels are the standard Fortran subroutines targeted at the CPU architectures. Finally, in the HIP backend, the Equation HIP further calls the wrappers required to execute the HIP kernels through C-Fortran interoperations.

3.2. HIP implementation

Our approach to the HIP implementation follows the structure used in the CUDA Fortran backend. This means that much of the CPU memory management, I/O routines and MPI implementation will remain largely the same and this is naturally achieved by exploiting the code separation guaranteed through the described object oriented approach. The majority of the changes for HIP come from the creation of the GPU memory

allocations, the wrapper interface and finally the HIP kernels written in C++. It should be noted that the API to access the HIP functionality from Fortran is done through HIPFort. The first fundamental change between CUDA Fortran and HIP is the creation of the GPU device arrays. For CUDA Fortran we create allocatable arrays with the device tag, but for HIP we create a pointer of a similar dimension. To better understand these differences, we provide an example of the creation and use of the *w_gpu* device array containing the conserved variables.

Listing 1 shows the main differences in creating device arrays using the CUDA Fortran and HIP approaches. Here *rkind* corresponds to *real64* or *real32* Fortran 2008 predefined precision types that can be chosen before compilation. All benchmarks reported in this paper refer to the double precision *real64* type.

CUDA Fortran

```
real(rkind), allocatable, dimension(:,:,:,:), device :: w_gpu
```

HIP

```
real(rkind), pointer, dimension(:,:,:,:) :: w_gpu
```

Listing 1: Creation of device arrays.

Listing 2 illustrates the allocation methods. In CUDA Fortran we use the standard *allocate* statement. For HIP we use the *hipMalloc* routine. The *hipcheck* routine which is implemented in HIPFort checks the return status of the passed APIs. This allocation of the main device array in the order (x, y, z, v), where x, y, z are the three spatial coordinates and v the conserved variables, guarantees CUDA/HIP thread coalescence when thread parallelization involves the first array dimension x. For the calculation of convective fluxes in the x direction, however, the first array dimension cannot be parallelized, and additional field arrays of different order (y, x, z, v) are allocated and filled at each iteration, transposing the normal field array. More details on this implementation can be found in [17].

CUDA Fortran

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

HIP

```
call hipCheck(hipMalloc(w_gpu, (nx+ng)-(1-ng)+1, (ny+ng)-(1-ng)+1, (nz+ng)-(1-ng)+1,nv))
```

Listing 2: Allocation of device arrays.

A corresponding CPU array, *w_cpu*, is used during the code initialization and finalization procedures. Before time evolution is initiated, the CPU array is transferred to the GPU device memory. Listing 3 illustrates the differences between the two approaches.

CUDA Fortran

```
w_gpu = w_cpu
```

HIP

```
call hipCheck(hipMemcpy(w_gpu,w_cpu,hipMemcpyHostToDevice))
```

Listing 3: CPU to GPU transfers.

For kernel implementation, CUDA Fortran employs two types of kernels. The most computationally intensive kernels are explicitly written with a block and grid configuration at call (global attribute kernels), while the relatively simple kernels are automatically generated using the kernel loop directive (*cuf* kernels) feature of CUDA Fortran. For HIP, however, all kernels must be written explicitly as there is no native support for directive-based methods. In addition to the main kernel, a wrapper on the HIP side and a call to this wrapper (with matching interface) on the Fortran side are required. In Listing 4 we take an example of a simple flux update kernel to explain the main differences. The CUDA Fortran version consists of a simple *cuf* kernel with directives placed before the loop and specifying the number of nested loops that are parallelized.

In the context of the HIP backend, we call a wrapper that contains the *hipLaunchKernelGGL* kernel launch and passes the corresponding C address object of the device arrays as an argument using Fortran's native *c_loc* procedure. Consequently, the implementation of this kernel is constituted by three components: a Fortran interface for the wrapper, a wrapper to launch the HIP kernel, and finally the HIP kernel. The details of this implementation are provided in Listing 5. The interface to call the HIP wrapper is created to specify the arguments and the data type that must be compatible for Fortran-C interoperability. On the HIP side, we have a wrapper function with arguments that match the interface on the Fortran side along with the grid and block dimensions needed for the kernel launch. The vector $(THREAD\_X, THREAD\_Y, THREAD\_Z)$ represents the block dimensions, which can be different for each kernel. This wrapper will eventually launch the HIP kernel, which will then perform the required operations. It should be noted that the first argument of the wrapper corresponds to the kernel stream. In this case, we use the default stream to launch the kernel, therefore, a *c_null_pointer* is passed from the Fortran side to enforce this.

Here, similar to the *rkind* parameter on the Fortran side, the *real* data type for the HIP kernel and wrapper can be a *float* or a *double*. In order to accommodate the discrepancies in the index ordering system between C++ and Fortran, we transform the multi-dimensional device arrays from

CUDA Fortran

```fortran
subroutine update_field_cuf(nx,ny,nz,ng,nv,w_gpu,fln_gpu)
  integer :: nx,ny,nz,nv,ng
  real(rkind), dimension(1-ng:,1-ng:,1-ng:,1:), intent(inout), device :: w_gpu
  real(rkind), dimension(1:,1:,1:,1:), intent(in), device :: fln_gpu
  integer :: i,j,k,m
  !$cuf kernel do(3) <<<*,*>>>
  do k=1,nz
    do j=1,ny
      do i=1,nx
        do m=1,nv
          w_gpu(i,j,k,m) = w_gpu(i,j,k,m)+fln_gpu(i,j,k,m)
        enddo
      enddo
    enddo
  enddo
endsubroutine update_field_cuf
```

HIP

```fortran
subroutine update_field_kernel(nx,ny,nz,ng,nv,w_gpu,fln_gpu)
  integer :: nx,ny,nz,nv,ng
  real(rkind), dimension(:,:,:,:), target :: w_gpu
  real(rkind), dimension(:,:,:,:), target :: fln_gpu
  call update_field_kernel_wrapper(c_null_ptr,nx,ny,nz,nv,ng,c_loc(w_gpu),c_loc(fln_gpu))
endsubroutine update_field_kernel
```

Listing 4: A general comparison of kernel subroutines between CUDA Fortran and HIP. The CUDA Fortran implementation uses a *cuf* directive for automatic kernel generation. The HIP implementation has a call to the wrapper which launches a HIP kernel.

Fortran into one-dimensional arrays on the C++ side by defining a custom macro for each GPU device array. By adopting this approach, we essentially use the same implementation and index ordering as the CUDA Fortran implementation while abstracting the C++-specific index system through the array index macro. Moreover, we believe that this approach will also enhance the readability of the HIP kernels, as the majority of the content remains consistent with the CUDA Fortran version. It is important to note that there is an additional cost associated with evaluating the index during runtime. However, this cost is justified by the benefits previously mentioned. For better clarity, a macro for the linearization of the *w_gpu* device array is provided in Listing 6.

Reduction operations are common in any CFD solver, and the strategy for porting reduction kernels is illustrated in the Listings 7 and 8. For CUDA Fortran, the implementation is relatively straightforward when using *cuf* directives, as the variable to be reduced is specified in the directive.

For the HIP implementation, similar to Listing 4, we specify a subroutine to call the kernel wrapper (see Listing 7). We employ the hipCUB [39] library, which is part of the ROCm software stack, to perform the required reductions. The reduction is performed in two steps. First, a kernel populates the device work array (*redn_3d_gpu*). Second, this array is passed to a hipCUB wrapper function for reduction (borrowed from [40]). All reduction operations in STREAmS-2 are performed on arrays of the same size, which allows us to reuse the same array for all reduction operations, thus limiting memory usage. Also, similar to the linearization of the *w_gpu*, we use the *_I3_REDN_3D* macro for the *redn_3d_gpu* array.

3.3. `sutils` - *an automatic porting tool*

The preceding section demonstrates that the CUDA Fortran and HIP kernels share several conceptual similarities, yet diverge significantly in their programmatic implementations. Nevertheless, they may adhere to a generalizable pattern under some constraints that could be automated through a source-to-source translation. This is the primary rationale behind `sutils` (an acronym for STREAmS utilities) and much of the subsequent development was a natural consequence.

`sutils`, developed in-house with a specific focus on STREAmS-2, is a tool capable of analyzing the main development branch based on CUDA Fortran and generating code for other backends. Originally developed to generate a CPU backend to supplement the CUDA Fortran backend, this paper presents how this tool has been further extended to support generation of a HIP backend. Consequently, the `sutils` tool is a source code generator providing code that is ready for compilation and has the potential to extend its support to other backend generation. To summarize, the tool is intended to achieve certain goals, which include:

- A guaranteed translation carried out in accordance with the specified code policies, which are consistently adhered to throughout the STREAmS-2 development process
- The Production of fully functional backends from the CUDA Fortran version of the code
- The Production of a perfectly readable code which can be easily modified if required

In addition to the HIP backend discussed in the present paper, `sutils` proved to be effective in the generation of other backends like OpenMP offload (see [41]).

Software design and features

`sutils` tool is written in Python 3 and makes extensive use of regular expressions to match and extract the underlying implementation details of CUDA Fortran kernels. The extracted information essentially contains the parsed information, which is stored in a Python dictionary. This dictionary will be used to perform the required translations and generate the desired backend. To help generate the kernels, a Python-based templating engine called Mako is used to setup static parts of the kernels. The aforementioned templates and dictionary are employed to construct the corresponding

(a) Interface:

```
interface
subroutine update_field_kernel_wrapper(stream,nx,ny,nz,nv,ng,w_gpu,fln_gpu)&
bind(c,name="update_field_kernel_wrapper")
 import :: c_ptr,c_int
 implicit none
 type(c_ptr), value :: stream
 integer(c_int), value :: nx,ny,nz,nv,ng
 type(c_ptr), value :: w_gpu,fln_gpu
end subroutine update_field_kernel_wrapper
end interface
```

(b) Wrapper:

```
extern "C"{
void update_field_kernel_wrapper(hipStream_t stream,int nx,int ny,int nz,int nv,int ng,
                       real *w_gpu,real *fln_gpu){
 dim3 block(THREAD_X,THREAD_Y,THREAD_Z);
 dim3 grid(divideAndRoundUp((nx)-(1)+1,block.x),
       divideAndRoundUp((ny)-(1)+1,block.y),
       divideAndRoundUp((nz)-(1)+1,block.z));

 hipLaunchKernelGGL((update_field_kernel),grid,block,0,stream,nx,ny,nz,nv,ng,
                       w_gpu,fln_gpu);
 }
}
```

(c) Kernel:

```
__global__ void update_field_kernel(int nx,int ny,int nz,int nv,int ng,
 real *w_gpu,real *fln_gpu){
 int i = 1+(threadIdx.x + blockIdx.x * blockDim.x);
 int j = 1+(threadIdx.y + blockIdx.y * blockDim.y);
 int k = 1+(threadIdx.z + blockIdx.z * blockDim.z);
 if(i <= nx && j <= ny && k <=nz){
  for(int m=1; m<nv+1; m++){
    w_gpu[__I4_W(i,j,k,m)] = w_gpu[__I4_W(i,j,k,m)]+fln_gpu[__I4_FLN(i,j,k,m)];
  }
 }
}
```

Listing 5: Specification of interface, wrapper and kernel for the update_flux kernel based on the HIP backend. *(a)* Interface contains definitions for all the arguments to be passed to the wrapper. *(b)* Wrapper contains the block and grid definitions with a kernel launcher. *(c)* Kernel with the parallel loop index definitions along with the main operations.

```
#define __I4_W(i,j,k,m) (((i)-(1-ng))+((nx+ng)-(1-ng)+1)*((j)-(1-ng))+
                ((nx+ng)-(1-ng)+1)*((ny+ng)-(1-ng)+1)*((k)-(1-ng))+
                ((nx+ng)-(1-ng)+1)*((ny+ng)-(1-ng)+1)*((nz+ng)-(1-ng)+1)*((m)-(1)))
```

Listing 6: Linearization of the multi-dimensional array *w_gpu* to one dimension.

backend code. Furthermore, an input file based on TOML file format is utilized to modify specific parameters of individual kernels, such as the number of parallel loops, grid/block dimensions, and launch bounds, within the generated backends.

sutils package contains two crucial classes: KernelExtract and BackendGenerate. The KernelExtract class (see Fig. 2) is primarily responsible for parsing the main backend-dependent portions of the CUDA Fortran code (see Fig. 1). As a preliminary step, all GPU device arrays are extracted to generate the index linearization macros which are stored in a dictionary, as illustrated in Listing 6. Furthermore, the implementation of the abstract class will extract the cuf directives, global and device kernels. The BackendGenerate abstract class (see Fig. 3) contains the KernelExtract object as one of its attributes. Its implementation of the abstract methods is carried out through the child classes, based on the user-chosen backend (CPU, HIP, or OpenMP). Additionally, the required static Mako templates are also set by the implemented classes. There are also further helper routines integrated into these abstract classes, depending on the chosen backend. To illustrate, in the case of HIP backend generation, a FortranToCpp procedure is developed to translate Fortran based kernels to C++.

CUDA Fortran to HIP translation

A translation from CUDA Fortran to HIP could potentially be accomplished through the utilization of a tool called GPUFORT [40]. This approach was initially contemplated, yet the focus subsequently shifted to the development of sutils. GPUFORT remains a noteworthy project, yet at the present moment, it is still a research tool. There were several reasons that led to the decision against the usage of GPUFORT in this particular work. At the time of development, we were unable to identify any significant Fortran codes that had successfully utilized GPUFORT to generate the HIP backend. The examples primarily addressed straightforward cases, and we were unable to reproduce them for the object-oriented software design of STREAmS-2. Furthermore, the generated code was overly generic, making readability and further modifications challenging. Finally, from the perspective of GPUFORT development, there have been no updates since the last commit in September 2021.

CUDA Fortran

```fortran
subroutine compute_residual_cuf(nx,ny,nz,ng,nv,fln_gpu,dt,residual_rhou)
  integer :: nx,ny,nz,ng,nv
  real(rkind), intent(out) :: residual_rhou
  real(rkind), intent(in) :: dt
  real(rkind), dimension(1:nx, 1:ny, 1:nz, nv), intent(in), device :: fln_gpu
  integer :: i,j,k
  residual_rhou = 0.0_rkind
  !$cuf kernel do(2) <<<*,*>>> 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
endsubroutine compute_residual_cuf
```

HIP

```fortran
subroutine compute_residual_kernel(nx,ny,nz,ng,nv,fln_gpu,dt,residual_rhou,redn_3d_gpu)
  integer :: nx,ny,nz,ng,nv
  real(rkind) :: dt
  real(rkind) :: residual_rhou
  real(rkind), dimension(:,:,:,:), target :: fln_gpu
  real(rkind), dimension(:,:,:), target :: redn_3d_gpu
  residual_rhou = 0.0_rkind
  call compute_residual_kernel_wrapper(c_null_ptr,nx,ny,nz,ng,nv,dt, &
                        c_loc(fln_gpu),residual_rhou,c_loc(redn_3d_gpu))
endsubroutine compute_residual_kernel
```

Listing 7: A general comparison of reduction procedures between CUDA Fortran and HIP. The CUDA Fortran implementation uses a *cuf* directive with the reduction variable specified in the directive.
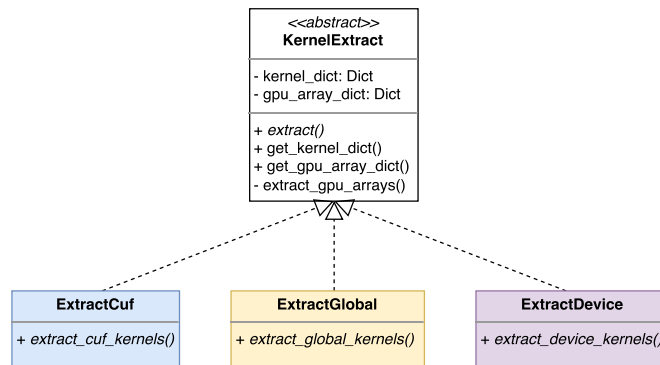

Fig. 2. UML diagram representing the structure of KernelExtract abstract class. The implementations are based on the three possible kernel definitions in CUDA Fortran: cuf directives, global kernels and device kernels.
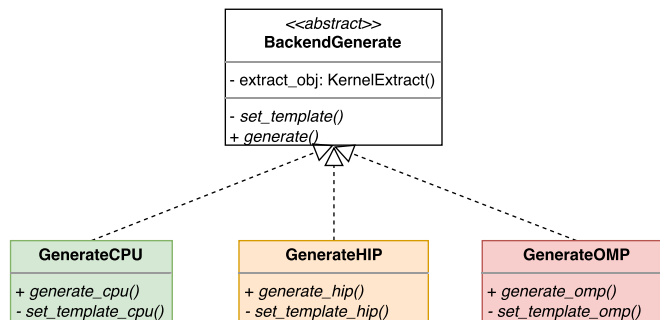
Fig. 3. UML diagram representing the structure of BackendGenerate abstract class. The implementation is based on the three possible backends that `sutils` can currently generate: CPU, HIP and OpenMP offload.

Conversion from the CUDA Fortran backend to HIP is more complex than obtaining the CPU backend because the HIP kernels are not written in Fortran and therefore, in addition to the kernels written in C++ we further require a wrapper and an interface to call it from the Fortran side as discussed in the previous section. Fig. 4 gives a very brief flowchart of how `sutils` would work in the HIP generation mode. As outlined earlier,

(a) Interface:

```
interface
subroutine compute_residual_kernel_wrapper(stream,nx,ny,nz,ng,nv,dt, &
                              fln_gpu,residual_rhou,redn_3d_gpu)&
bind(c,name="compute_residual_kernel_wrapper")
 import :: c_ptr,c_int,c_rkind
 implicit none
 type(c_ptr), value :: stream
 integer(c_int), value :: nx,ny,nz,ng,nv
 real(c_rkind), value :: dt
 type(c_ptr), value :: fln_gpu
 real(c_rkind) :: residual_rhou
 type(c_ptr), value :: redn_3d_gpu
end subroutine compute_residual_kernel_wrapper
end interface
```

(b) Wrapper:

```
extern "C"{
void compute_residual_kernel_wrapper(hipStream_t stream,int nx,int ny,int nz,int ng,int nv,
                            real dt,real *fln_gpu,
                            real *residual_rhou,real *redn_3d_gpu){
 dim3 block(THREAD_X,THREAD_Y,THREAD_Z);
 dim3 grid(divideAndRoundUp((nx)-(1)+1,block.x),
        divideAndRoundUp((ny)-(1)+1,block.y),
        divideAndRoundUp((nz)-(1)+1,block.z));

 hipLaunchKernelGGL((compute_residual_kernel),grid,block,0,stream,nx,ny,nz,ng,nv,dt,
                            fln_gpu,redn_3d_gpu);
 // hipCUB reduction for sum
 reduce<real, reduce_op_add>(redn_3d_gpu, nz*ny*nx, residual_rhou);

}
```

(c) Kernel:

```
__global__ void compute_residual_kernel_residual_rhou(int nx,int ny,
int nz,int ng,int nv,real dt,real *fln_gpu,real *redn_3d_gpu){
 int i = 1+(threadIdx.x + blockIdx.x * blockDim.x);
 int j = 1+(threadIdx.y + blockIdx.y * blockDim.y);
 int k = 1+(threadIdx.z + blockIdx.z * blockDim.z);
 if(i <= nx && j <= ny && k <=nz){
  redn_3d_gpu[__I3_REDN_3D(i,j,k)] = 0.0;
  redn_3d_gpu[__I3_REDN_3D(i,j,k)] = ((fln_gpu[__I4_FLN(i,j,k,2)]/dt))*
                        ((fln_gpu[__I4_FLN(i,j,k,2)]/dt));
 }
}
```

Listing 8: Specification of Interface, wrapper and kernel for the compute_residual_kernel based on the HIP backend. *(a)* Interface contains definitions for all the arguments to be passed to the wrapper. *(b)* Wrapper contains the block and grid definitions with a kernel launcher followed by the hipCUB reduction routine for summation. *(c)* Kernel mainly populates the reduction array which will be reduced by hipCUB.

for a chosen background the tool first parses the CUDA Fortran code and writes all the necessary information to a Python dictionary. In the context of HIP porting, the tools main objectives are to use this dictionary and perform three main tasks (represented by the red blocks in the flowchart),

- Translate the kernel from Fortran to C++ and pass the translated code through the mako template and generate the kernels
- Generate the interfaces and calls to wrapper in Fortran
- Replace the GPU array creation, allocation and transfers (as in Listings 1, 2, 3)

To expand more on this, we consider the example of the update_field_cuf kernel which was used to describe the HIP implementation in the previous section. Listing 9 gives a truncated representation of how the kernel dictionary looks like for this kernel. There are several other information parsed and stored in this dictionary which is not shown due to space constraints. This piece of information is common to all other backend generation. In the next step, we translate the kernel from Fortran to C++ and pass this information along with the kernel dictionary to the Mako template (as in 4). Listing 10 represents the Mako template to generate a global HIP kernel which does not contain any reduction operations. This generates the HIP kernel and wrapper shown in Listings 5b and 5c.

sutils tool, in its basic form, is capable of translating the CUDA Fortran backend without any errors. However, to produce optimized code, sutils can also perform changes to the kernel at a higher level by specifying an input TOML file. Through this TOML file, the user has some control over changing some of the contents of the parsed dictionary (see Listing 9). These include the number of parallel loops (variable *num_loop*), loop index ordering (variables *idx* and *idx_range*), and others. This is done to address situations where the CUDA Fortran kernel parameter configurations are not optimal for the other generated GPU backends. The TOML input file for the reduction kernel compute_residual_cuf, as specified in Listing 11, is presented in Listing 8. In this case, the CUDA Fortran cuf directive had the two outer loops parallelized. In contrast, in the HIP case, by specifying num_loop = 3 in the input file, the translator can automatically adjust the output code with all the three loops parallelized. This input file thus allows us to test several parameters in the HIP kernel in order to identify the optimal setup.
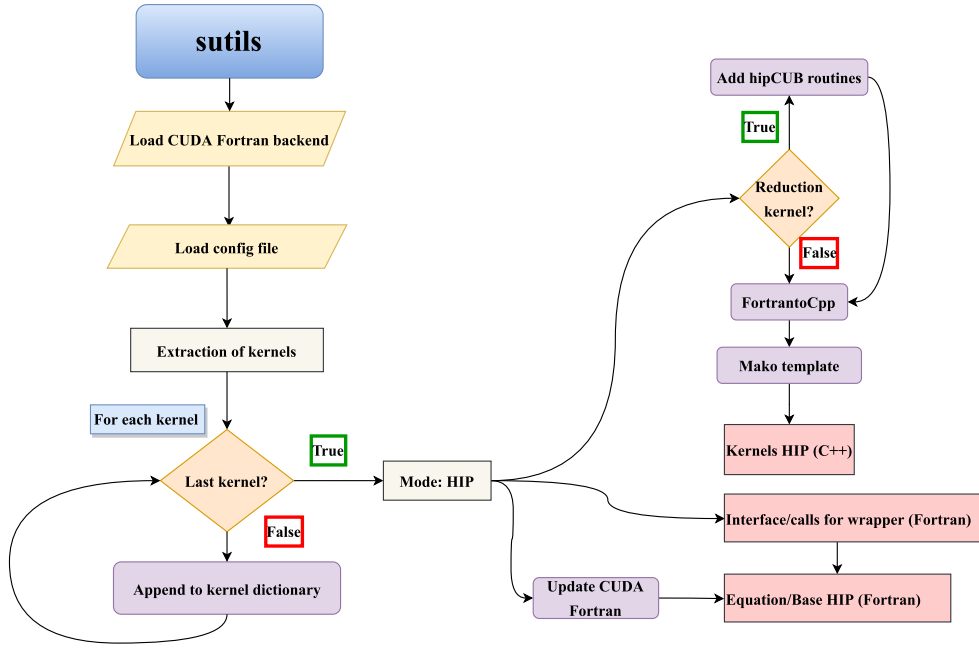
Fig. 4. Program flow of `sutils` to obtain the HIP backend from the CUDA Fortran backend.

```
"update_field_cuf": {
 "subroutine_info": {
  "subroutine_name": "update_field_cuf",
  "subroutine_decl": "subroutine update_field_cuf(nx, ny, nz, ng, nv, w_gpu, fln_gpu)\n",
  "subroutine_args": ["nx", "ny", "nz", "ng", "nv", "w_gpu", "fln_gpu"],
  "subroutine_variables": "integer :: nx, ny, nz, nv, ng\nreal(rkind), dimension(1-ng:,1-ng:,1-ng:,1:), intent(inout), device
        :: w_gpu\nreal(rkind), dimension(1:,1:,1:,1:), intent(in), device :: fln_gpu\ninteger :: i,j,k,m,iercuda\n",
  "subroutine_code": "\n!$cuf kernel do(3) <<<*,*>>>\ndo k=1,nz\ndo j=1,ny\ndo i=1,nx\ndo m=1,nv\nw_gpu(i,j,k,m) = w_gpu(i,j,k
        ,m)+fln_gpu(i,j,k,m)\nenddo\nenddo\nenddo\nenddo\n!@cuf iercuda=cudadevicesynchronize()\n"
 },
 "kernel_info": {
  "kernel": [
    "!$cuf kernel do(3) <<<*,*>>>\ndo k=1,nz\ndo j=1,ny\ndo i=1,nx\ndo m=1,nv\nw_gpu(i,j,k,m) = w_gpu(i,j,k,m)+fln_gpu(i,j,k,m)
        \nenddo\nenddo\nenddo\nenddo\n"
  ],
  "serial": [
    "do m=1,nv\nw_gpu(i,j,k,m) = w_gpu(i,j,k,m)+fln_gpu(i,j,k,m)\nenddo"
  ],
  "num_loop": [3],
  "idx": [
    ["k", "j", "i"]
  ],
  "idx_range": [
    [
      ["1", "nz"],
      ["1", "ny"],
      ["1", "nx"]
    ]
  ],
  "cuf_directive": ["!$cuf kernel do(3) <<<*,*>>>"]
 },
 "var_info": {
  "real_arrays": ["w_gpu", "fln_gpu" ],
  "integer" : ["nx" , "ny" , "nz", "nv", "ng" ],
  "linteger" : ["i" , "j" , "k" , "m" , "iercuda"]
 }
}
```

Listing 9: Truncated kernel dictionary for the update_field_cuf kernel.

Limitations

While the `sutils` tool is effective in porting STREAmS-2 to multiple backends, there are some limitations that require consideration. As previously stated, the tool is capable of performing rudimentary optimizations through kernel parameters. However, more complex optimizations tailored to CUDA Fortran cannot be directly translated to other backends. Similarly, newer strategies suited to a particular backend cannot be currently

```
__global__ void ${launch_bounds} ${kernel_name}(${kernel_args}){
//Kernel for ${kernel_name}

  ${local_variables}

  ${thread_declaration}

  if(${loop_conditions}){
    ${translated_kernel}
  }
}

extern "C"{
 void ${wrapper_name}(hipStream_t stream,${kernel_args}){
   dim3 block(${block_definition});
   dim3 grid(${grid_definition});

   hipLaunchKernelGGL((${kernel_name}),grid,block,0,stream,${wrapper_args});

 }
}
```

Listing 10: Mako template for the generation of global HIP kernels.

```
[compute_residual_cuf]
[compute_residual_cuf.hip]
num_loop = 3
```

Listing 11: Input TOML file for kernel optimization.

implemented without manual intervention. Another limitation is that the testing strategy employed in the tool is in its infancy. The correctness of the results obtained from the ported code is also manually verified through test simulations. Nevertheless, an automated testing framework for both the tool and the solver is currently under development, which will undoubtedly enhance the tool's reliability and eliminate potential defects.

4. Performance analysis

In this section we present the performance results of STREAmS-2 for the CUDA Fortran and HIP backends. The section is divided into two parts. In the first part, we evaluate the performance based on a single GPU card. The single-GPU performance analysis represents the fundamental evaluation of the overall code architecture – in particular the memory layout – and the computational kernels: different numerical conditions are evaluated against each other and ultimately against the peak performance of the hardware used. To achieve this, we use NVIDIA A100 and AMD MI250x GPUs, which form the computational core of several modern supercomputers. However, large production runs typically require codes to be run on multiple nodes with many GPUs. In the second part, we look at the scalability performance of the CUDA Fortran and HIP backends of STREAmS-2 based on two pre-exascale clusters that are part of the EuroHPC JU [42].

4.1. Single-GPU evaluation

We evaluate the performance of STREAmS-2 on single-GPU architectures and for a reference CPU configuration. For the CPU architecture, we consider a full compute node (2x AMD CPUs EPYC 7763) of the CPU partition of LUMI using MPI parallelization. This choice not only aims to achieve comparable times between CPUs and GPUs, but also to ignore any intra-node CPU effects and to provide comparisons between independent units. Table 1 describes the main features of the selected GPUs. Theoretical peak performance values are reported. The memory bandwidth is expected to play a crucial role given the explicit characterization of STREAmS solver algorithm. However, actual values differ from theoretical peak values and it is worth considering a more realistic estimation of bandwidth. To this aim, both CUDA and HIP official repositories include an example simple code to estimate bandwidth in more realistic conditions. The measured bandwidths for A100 GPU and MI250X GCD are reported in Table 2: both values are around 75% of theoretical peak ones with a small but measurable advantage of AMD GPU.

For the GPU tests, we consider a single card for NVIDIA V100, NVIDIA A100 and AMD MI100 GPUs. For AMD MI250x, we choose one of the two Graphics Compute Dies (GCDs) that make up a GPU card. Actually, the two GCDs are managed by two MPI processes in STREAmS-2 and the scheduler also identifies them as two GPU units.

The test case is based on a computational grid ($420 \times 250 \times 320$) which nearly saturates the memory availability of the V100 GPUs (16 GiB) that are part of the Marconi100 cluster. Results are reported in (Fig. 5). All simulations use a 6th order central scheme and a 5th order WENO scheme, which are generally adopted for production runs. Additional discussion of the impact of different orders of accuracy on run time will be provided later.

The results show that GPUs are generally much faster than a single full CPU node on LUMI. In particular, GPU times range from two to four times faster than the CPU configuration under consideration. The trends in GPU results unsurprisingly follow their release dates (for the same vendor) and peak predictions. However, later in this section we will see that the real performance does not strictly follow the expected predictions as we try

Table 1

Description of the GPUs used in single-GPU performance evaluations.

Vendor	NVIDIA	NVIDIA	AMD	AMD
Model	V100 GPU	A100 GPU	MI100 GPU	MI250x GCD
Release year	2017	2020	2020	2021
Memory (GiB)	16	40	32	64
Peak FP64 performance (TFLOPS)	7.00	9.70	11.50	23.90
Peak bandwidth (GB/s)	900.00	1555.00	1229.00	1630.00
Compiler	HPC-SDK 22.11 CUDA 11.0	HPC-SDK 22.11 CUDA 11.0	GCC 8.5.0 ROCm 4.5.2	GCC 11.2.0 ROCm 5.0.2
Profiling	NVIDIA Nsight Systems	NVIDIA Nsight Systems	rocprof	rocprof

Table 2

Memory bandwidth for A100 GPU and MI250X GCD: comparison of peak values and real values extracted using simple code from CUDA/HIP examples.

	Peak bandwidth	CUDA/HIP example code bandwidth	Real/peak
A100 GPU	1555 GB/S	1160 GB/S	75%
MI250X GCD	1630 GB/S	1282 GB/S	79%
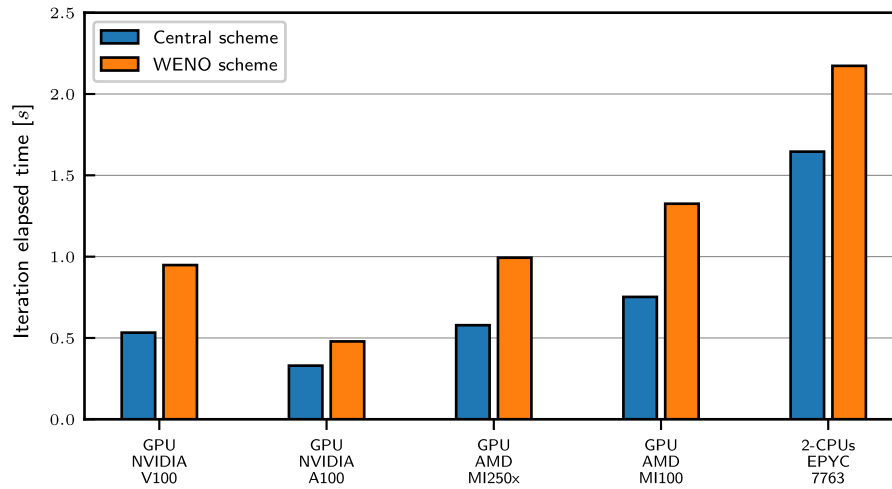
Fig. 5. Elapsed time per iteration (s) for STREAmS-2 using different HPC architectures for grid: $(420 \times 250 \times 320)$. Comparisons between central and WENO flux evaluation schemes. CPU average time is based on a full node of 128 cores.

to understand the reasons for the observed discrepancies. For example, the performance of the MI250x GCD is similar to that of the old generation V100 GPU, although the peak performance is remarkably different. We note that the time per iteration of the code in WENO mode is about 70% larger than the time in central mode considering the V100 GPU, MI100 GPU, and MI250X GCD. The gap narrows to around 45% for the A100 GPU and around 30% for the CPU.

We now look at the individual kernel performance across the GPU architectures (Fig. 6). Table 3 provides a description of the kernels used in the comparison. The results are reported in seconds unless otherwise stated in the figure. We have considered kernels based on three types that best represent the solver. The most computationally intensive kernels corresponding to convective and viscous flux evaluations (euler_x_central, euler_x_weno, euler_y_central, euler_y_weno, euler_z_central, euler_z_weno and visflx_nosensor). The convective flux kernels represented by the central and WENO schemes are shown here as two different kernels (in each grid direction). In principle, however, the solver can be run in single (central and WENO) or hybrid mode. The effects of these modes are discussed in detail later in this section. These kernels are characterized by high complexity, especially for the WENO branch with high register usage, which can be crucial in limiting GPU occupancy and the final performance results. It should also be noted that the convective flux evaluations are explicitly implemented as global kernels for CUDA Fortran due to the use of local private arrays and device functions, but the viscous fluxes (visflx_nosensor kernel) are implemented through *cuf* directives in the CUDA Fortran backend. We also include some kernels that are not particularly demanding, but interesting for our implementation strategy and performance evaluations. These include a basic linear algebra kernel, update_flux, and a transposition kernel, euler_x_transp. Finally, reduction kernels: compute_residual (simple sum reduction), compute_dt (max reduction) and force_rhs_1 (a more complex sum reduction) are used for the comparisons.

For convective flux kernels, performance is again comparable between the V100 GPU and the MI250x GCD, with the A100 GPU outperforming in all cases. The A100 GPU has an advantage of about 50% for WENO executions and about 35% for central executions. As the central execution is closer to basic linear algebra computation (compared to the more complex WENO code pattern), the results are expected to closely follow the peak bandwidth trends. Indeed, the update_flux kernel, which is very similar to a simple matrix addition, shows close performance results for all devices with an average deviation of around 10%, although the A100 GPU still outperforms the MI250x GCD despite the fact that the peak bandwidth should (even slightly) favor the latter. A possible explanation of this behavior is given by the index linearization cost which is present for our HIP
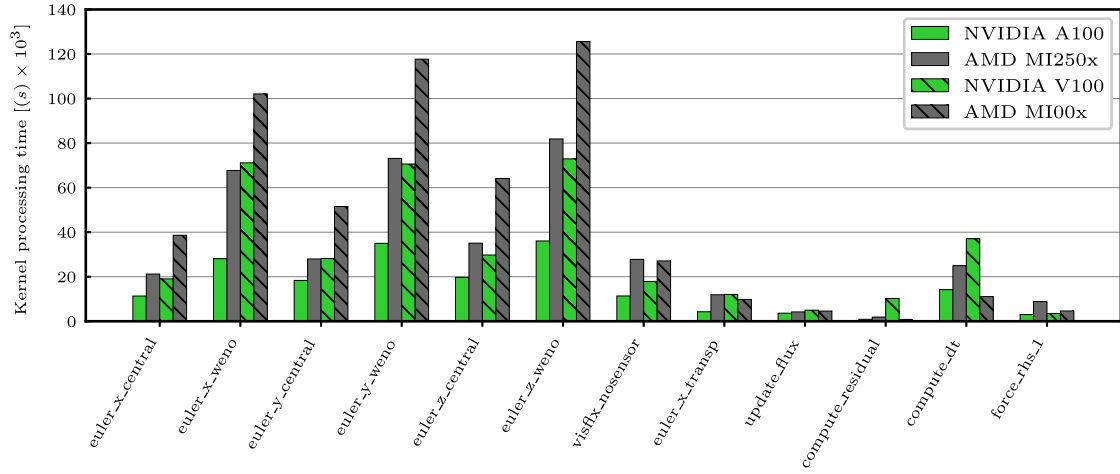
Fig. 6. Kernel processing time ($s \times 10^3$) comparison for significant kernels of STREAmS-2 across four GPU architectures for grid: ($420 \times 250 \times 320$).

Table 3
Description of the kernels used in Fig. 6. CUDA Fortran employs global and cuf-directive kernels. HIP only employs global kernels with hipCUB implementation for reductions.

Kernel	Kernel type		Description
	CUDA Fortran	HIP	
euler_x_central	Global	Global	Convective flux evaluation in x-direction using Central scheme
euler_x_weno	Global	Global	Convective flux evaluation in x-direction using WENO scheme
euler_y_central	Global	Global	Convective flux evaluation in y-direction using Central scheme
euler_y_weno	Global	Global	Convective flux evaluation in y-direction using WENO scheme
euler_z_central	Global	Global	Convective flux evaluation in z-direction using Central scheme
euler_z_weno	Global	Global	Convective flux evaluation in z-direction using WENO scheme
visflx_nosensor	cuf	Global	Viscous flux evaluation
euler_x_transp	cuf	Global	Transposition of main device array
update_flux	cuf	Global	Flux array update
compute_residual	cuf	Global hipCUB	Residual evaluation (sum reduction)
compute_dt	cuf	Global hipCUB	Time step evaluation (max reduction)
force_rhs_1	cuf	Global hipCUB	Forcing term evaluations (sum reduction)

implementation, as discussed in the previous Section. The transposition kernel results are somewhat unexpected; while the V100 GPU and MI250x GCD results are almost identical, the MI100 GPU results are slightly better and, more interestingly, the A100 GPU results are about three times better. Optimal transposition kernels would require extensive use of shared memory and more advanced techniques. While *cuf* directive kernels can automatically produce such optimized code, the reason for such a large difference between V100 and A100 GPUs is unclear. For the AMD counterparts, we decided to implement only naive transposition to maintain code clarity and simplify automation, as the impact of transposition on global run times is negligible. The results for reduction kernels (compute_residual, etc.) are a little more difficult to analyze. While the superiority of the A100 GPU always holds, the results for other GPUs are more variable, but this is to be expected as different reduction implementations are involved. However, the only reduction kernel that is computationally relevant, compute_dt, is typically called once every 10 iterations in production runs, so its contribution to the total run time is negligible. The last kernel of interest, visflx_nosensor, although structurally simple (mainly derivatives and linear combinations of matrices), contains a fairly large number of lines (around 200). The performance advantage of NVIDIA is more evident in this case for all GPU generations. The results are analyzed in more detail in the following sections.

To evaluate the behavior of STREAmS-2 on different architectures, we perform some targeted tests. From here on, we only consider the A100 GPU and the MI250x GCD, which are more prevalent in HPC clusters at the time of writing. We start by measuring the impact of launch bounds on kernel

Table 4

Effect of Launch bounds for grid: $(420 \times 250 \times 320)$. All launch bounds are taken as multiples of 128. Both the kernel processing time of euler_x, euler_y and euler_z kernels and iteration elapsed time in $(s \times 10^3)$.

Launch bounds	euler_x		euler_y		euler_z		Iteration elapsed time	
	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD
	Time(s) $\times 10^3$							
Central scheme								
128	11.2	23.9	18.2	32.0	19.5	41.2	327.6	618.2
256	11.3	23.6	18.3	31.4	19.8	39.2	331.1	611.3
384	10.9	21.1	16.4	27.9	18.0	35.1	317.5	579.3
512	12.6	23.8	15.9	30.9	18.0	38.5	321.5	606.8
640	16.4	23.8	17.6	30.9	17.5	38.5	336.2	606.1
WENO Scheme								
128	27.8	71.8	34.7	71.4	35.8	75.6	476.8	984.2
256	27.9	72.0	34.9	70.4	35.8	75.3	477.8	982.7
384	28.3	67.7	32.2	73.1	37.7	81.4	476.4	994.5
512	36.8	73.4	39.7	71.0	43.5	81.0	542.7	100.4
640	50.5	122.3	48.0	92.2	46.2	87.6	616.0	1234.1

Table 5

Effect of the number of parallel loops for grid: $(420 \times 250 \times 320)$. Kernel processing times for update_flux and visflx_nosensor kernels in $(s \times 10^3)$.

Parallel loops	update_flux		visflx_nosensor	
	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD
	Time(s) $\times 10^3$			
2	3.3	6.1	11.4	53.6
3	3.1	4.2	16.0	27.8

run time (see Table 4). Convective kernels are characterized by high complexity and require a large number of registers. Under these conditions, fine-tuning the thread block can be effective in optimizing execution. This is particularly true because launch bounds can be specified in the kernel definition itself to help the compiler decide on the number of registers that can be safely allocated. We consider the three most computationally demanding kernels and obtain the kernel and elapsed time for different thread block configurations and corresponding launch_bounds declarations. We vary the thread configurations along y, while maintaining 128 threads for the x block size. A separate test was performed with multiples of 64, but the performance was either similar or lower and is therefore not reported here. Note that since many parallelized loops span only two directions, we always fix the number of threads along z as one.

For the central scheme, both the A100 GPU and the MI250x GCD have their lowest final run times for the 128×3 configuration, which is also reflected in the individual kernel run times. However, for the WENO scheme, the A100 GPU performs better with 128×3 and the MI250x GCD performs better with 128×2. For all further analysis in this section, we stick with the 128×3 configuration for two main reasons:

- The difference between the configurations for the WENO scheme is less than 1% as opposed to the central scheme where it is between 4-5%
- Production runs of STREAmS-2 generally involve the use of hybrid schemes, and in most cases the central scheme forms the bulk of the kernel runs

In the next test we look at the effect of the number of parallel loops on the kernel run time (see Table 5). This drastically affects the number of CUDA/HIP threads invoked by the kernel and therefore the number of operations performed by each thread. To illustrate this concept, we consider two kernels, update_flux and visflx_nosensor. As explained, the former is a minimal linear algebra kernel, while the latter is still structurally simple (no device function, no private local array) but has a much higher number of operations and consequently higher register usage.

For the CUDA Fortran backend, using *cuf* kernel directives, the number of parallel loops has minimal impact on the update_flux kernel. For visflx_nosensor kernel, however, using only two parallel loops significantly improves the timing probably due to pipelining intra-kernel parallelization and/or cache effects. On the other hand, for the HIP backend, parallelizing all three loop indices leads to a dramatic improvement in performance, probably because the AMD compiler and devices are not able to take advantage of significant intra-kernel optimizations. Predicting the optimal number of parallelized loops in general is not straightforward and probably requires manual attempts.

Next, we test the effect of thread masking on the kernel run times of the three convective kernels (see Table 6). It is known that if the number of grid points is not a multiple of the thread block, we will have inactive threads in the warp (in the case of NVIDIA) or wavefront (in the case of AMD). Depending on the extent of the inactive threads, we can expect a reduction in performance. We measure the grind time as the comparisons are on different, albeit slightly different grids. The grind time (T_g), also known as the data processing rate, is the ratio of the run time and the number of GPUs (N_{GPU}) to the number of grid points (N) and is expected to be constant under ideal conditions. Mathematically, the grind time can be obtained as $T_g = T \times N_{GPU}/N$. Intuitively, this can be thought of as the time it takes one GPU to process one grid point. In the Table 6, the grind time is given as the time to process one hundred million points. In the context of single GPU evaluations, $N_{GPU} = 1$, therefore we calculate the grind time as $T_g = T \times 10^8/N$. In this test, we always use the optimal thread configuration of 128×3 (see Table 4). To test thread masking, we start by considering grids that are multiples of 128 (the largest CUDA/HIP thread size) and gradually modify these numbers to reproduce less ideal conditions:

- Grid 1: $(384 \times 256 \times 384)$ - Multiples of 128 in all three directions

Table 6

Effect of thread masking. Individual kernel times for euler_x, euler_y and euler_z kernels are in $(s \times 10^3)$.

Grids	euler_x		euler_y		euler_z		Grind time (T_g)	
	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD	A100 GPU	MI250x GCD
	Time(s) $\times 10^3$							
Central scheme								
Grid 1	26.1	13.3	18.7	30.6	19.5	38.4	0.91	1.72
Grid 2	25.8	13.0	15.9	29.8	17.8	33.8	0.91	1.73
Grid 3	28.2	14.4	20.8	33.2	21.9	39.8	1.09	1.96
Grid 4	21.1	11.2	18.3	27.9	19.8	35.1	0.98	1.72
WENO scheme								
Grid 1	66.1	31.0	36.7	73.4	37.2	78.0	1.34	2.69
Grid 2	61.2	30.3	31.2	67.4	33.8	70.7	1.33	2.69
Grid 3	66.9	33.6	40.0	86.4	40.7	92.3	1.59	3.21
Grid 4	67.7	27.5	34.9	73.1	36.0	81.3	1.42	2.96

(a)   (b)
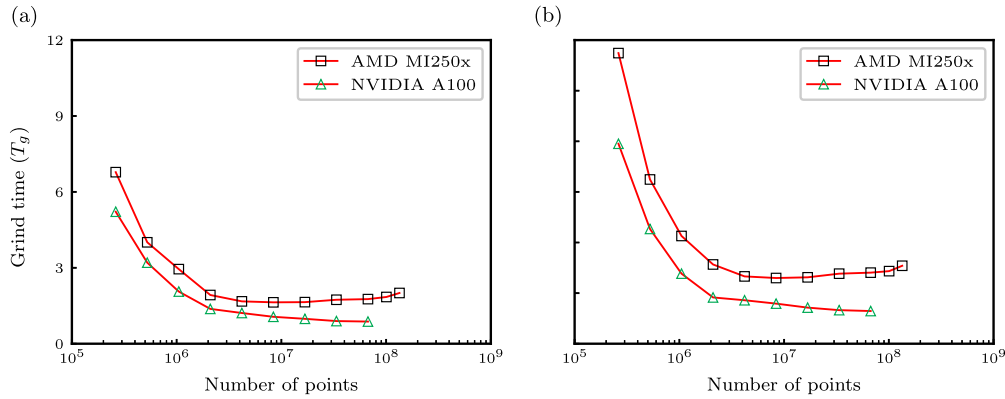
Fig. 7. Effect of grid size. *(a)* Central scheme. *(b)* WENO scheme.

- Grid 2: ($384 \times 256 \times 350$) - Multiples of 128 in two directions (x and y)
- Grid 3: ($420 \times 256 \times 350$) - Multiples of 128 in one direction (y)
- Grid 4: ($420 \times 250 \times 320$) - Not a multiple of 128 in any direction

From the Table 6, as expected, grids 1 and 2 give the best times in most cases. For the A100 GPU, the reduction in performance between best and worst case is around 20% for both the central and WENO schemes. For the MI250x GCD, the reduction is around 14% for the central scheme and 20% for the WENO scheme. Unexpectedly, the performance of grid 4 is generally better than grid 3. Overall, the use of a good thread block configuration is recommended for production runs where possible.

It is well known that GPUs perform well when processing large numbers of grid points. This is due to their inherent data parallelism. It is therefore interesting to measure performance in terms of the number of grid points, as this is a hard limit to strong scaling (where each GPU processes a progressively smaller part of the grid). Of course, since we are comparing different grids, we need to look at grind times (T_g) rather than total execution times. We use a base grid of around 0.25 million points and gradually double the grid size, with the largest grid being around 134 million points for the MI250x GCD and 67 million points for the A100 GPU.

Figs. 7a and 7b illustrate the effect of grid size for the central and WENO schemes respectively. The NVIDIA cards show a monotonous behavior as the number of points increases. We can see two different ranges if we consider the logarithmic scale. In the first range, from 0.25 to 1 million points for the central scheme and from 0.25 to 2 million points for the WENO scheme, the reduction in grind time is enormous, proving that this range of points is not enough to exploit the parallelization potential of the GPU. From 2 million grid points, there is still an improvement in grind time, but at a much lower rate. These two ranges are clearly visible for both central and WENO activation. All in all, the GPU delivers good results when the number of points to be processed exceeds 2 million. On the other hand, AMD's results show two trends: the first (number of points less than 2 million) is characterized by a sharp reduction in grind time, as is the case with its NVIDIA counterpart. The second, on the other hand, shows a slight increase in grind time. It turns out that there is an optimal number of grid points to obtain the best results on AMD cards, and it is not the largest possible case. The presented analysis also shows that the grid size chosen for most of the tests in this section is in the optimal range for the GPUs in question.

We performed a roofline analysis based on [43]. Roofline analysis is a very useful method to understand how the achieved performance compares to the peak performance of a device. It can also be used to identify bottlenecks, which can help with further optimizations. To obtain a roofline plot, the performance of a kernel is plotted against the Arithmetic Intensity (AI). AI is the ratio of floating point operations performed to data movement (FLOPs/Byte). Intuitively, a higher AI could represent better data locality. A roofline plot consists of two ceilings and a ridge. The region below the first ceiling, at lower AI, is limited by bandwidth, also known as the memory bounded region. Different memory layers could be considered, assuming data is found on High Bandwidth Memory (HBM), L2 cache and L1 cache, resulting in different memory ceilings. We will consider a standard roofline analysis based on HBM. The region below the second ceiling is limited by the peak performance of the device, also known as the compute bound region. Peak performance generally assumes that all operations are performed as Fused Multiply-Add (FMA), which is practically impossible to achieve for a real code. The ridge point is the transition point between the two ceilings.
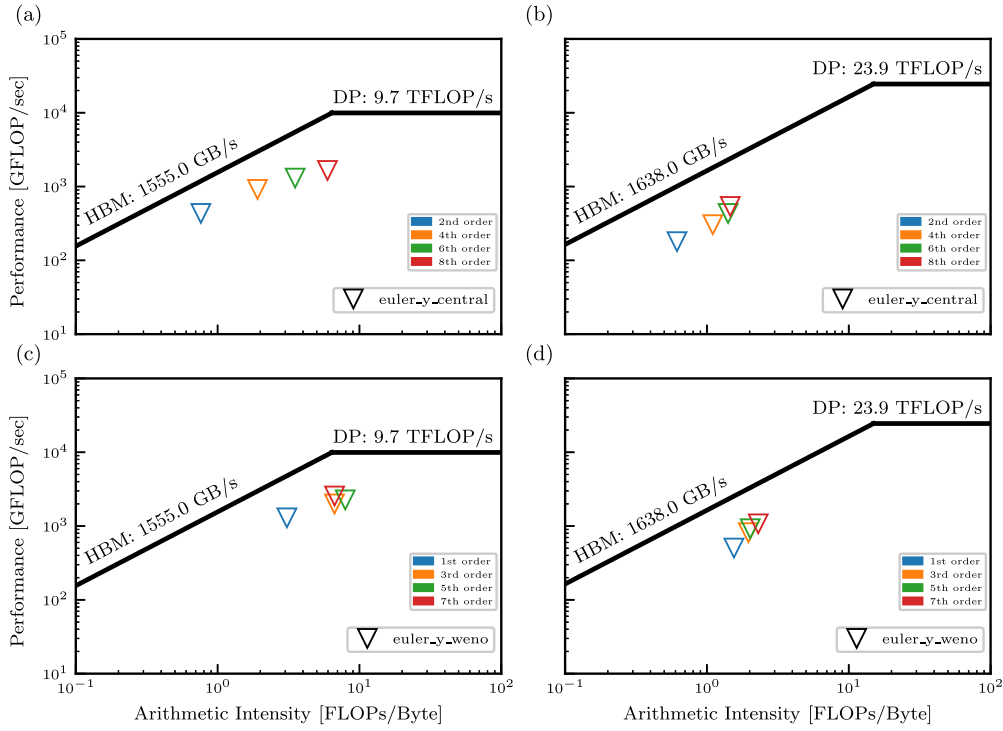
Fig. 8. Roofline analysis for euler_y_weno and euler_y_central kernels for grid: $(420 \times 250 \times 320)$. *(a)* and *(c)* AMD MI250x GCD. *(b)* and *(d)* NVIDIA A100 GPU. The marker corresponds to the kernel of the convective scheme, colored according to its order of accuracy.

Figs. 8a-8d illustrate the roofline plots for the euler_y kernel. Figs. 8a and 8c show the analysis for the central and WENO schemes based on the A100 GPU, respectively, while Figs. 8b and 8d show the same for the MI250x GCD. The plots also include comparisons between four different orders of accuracy for the central $(2^{nd}, 4^{th}, 6^{th}, 8^{th})$ and WENO $(1^{st}, 3^{rd}, 5^{th}, 7^{th})$ schemes.

As expected for an explicit CFD code, we are generally in the memory bound regions. However, there are important differences. The MI250x GCD is always memory bound for both kernels and all orders of accuracy, whereas the A100 GPU approaches the computational bound regions at high orders of accuracy. For the A100 GPU, the increase in computational intensity becomes more pronounced as the order of accuracy increases. This is particularly true for the central schemes. WENO schemes, on the other hand, show relatively high AI for the A100 GPU at every order. The MI250x GCD, on the other hand, is always limited to an AI range of 1 to 3 and the corresponding performance is very limited. This may be related to the reduced data reuse (this is explained in more detail in Fig. 9). As expected for realistic scientific codes, all points follow the behavior of the roofline boundary, but are a little far from the ceiling. This means that there is potential room to approach the hardware limits. However, it should be noted that the floating-point limits take into account full FMA operations, which are unrealistically achievable for STREAmS-2. In terms of HBM measurements, caching generally dramatically improves the perceived memory bandwidth. In other words, from the programmer's point of view (memory accesses measured against source code), the bandwidth is significantly greater than the measured HBM bandwidth. In this context, possible future work could include additional memory optimizations (e.g., shared memory). However, it should be noted that STREAmS-2, as a community code in perpetual evolution, has to ensure reasonable readability and maintainability, even considering the possible addition of new numerics. Therefore, it is not always practical to develop a solver that is too machine specific or has complex optimizations.

To better understand the effect of the order of accuracy, we now look separately at elapsed times, total data movement to and from HBM and the number of operations performed. From a developer's point of view, these quantities are usually measured per second, as this is more meaningful for analyzing performance. In this case, however, the quantities considered can be used to determine the capabilities of both the compilers and the caching mechanisms of the devices, which makes them interesting to study. The results are given for the total iteration time and for the euler_y kernel at different numerical accuracies, considering both central and WENO modes, and for both GPUs. The bottom x-axis represents the order of accuracy for the discretization of the convective central scheme (prefix C) and the viscous terms (prefix V), and the top x-axis represents the order of accuracy for the WENO reconstruction (prefix W). For example, W1 and C2 correspond to a first-order (i.e. upwind) WENO reconstruction and a second-order central scheme, respectively. Fig. 9a shows the iteration time of the solver with respect to the order of accuracy, while Fig. 9b shows the same for the euler_y kernel. Figs. 9c and 9d show the movement of the HBM data and the total operations performed by the kernel as a function of the order of accuracy.

Fig. 9a shows a substantial linear behavior for both GPUs and both central/WENO activation. Time ratios among different codes are similar but show a small advantage for A100 GPU runs when increasing the accuracy order. For instance, at first order WENO, both AMD and NVIDIA times are much closer, but this progressively increases with the order of accuracy. To simplify the analysis, we look at the run times for a single euler_y kernel. In Fig. 9b we see an increase in kernel time for the euler_y kernel that is virtually linear with increasing order of accuracy. As already discussed, for the higher orders of accuracy, the absolute values are always significantly different comparing the A100 GPU and the MI250x GCD, with a large advantage for the NVIDIA card. The other plots in this figure allow us to understand the possible reasons for this difference, despite the fact that AMD has higher peak performance values. In Fig. 9c we see that the data movement for the MI250x GCD is dramatically higher than for the A100 GPU in both cases. This is the main reason why NVIDIA shows a larger computational intensity as shown in the roofline analysis. As we only measure global memory accesses, this means that the NVIDIA GPU can extract the required data from registers and cache more often than its AMD counterpart. This is particularly true for the central scheme, which requires the same amount of data movement for each numerical order using the A100 GPU, while
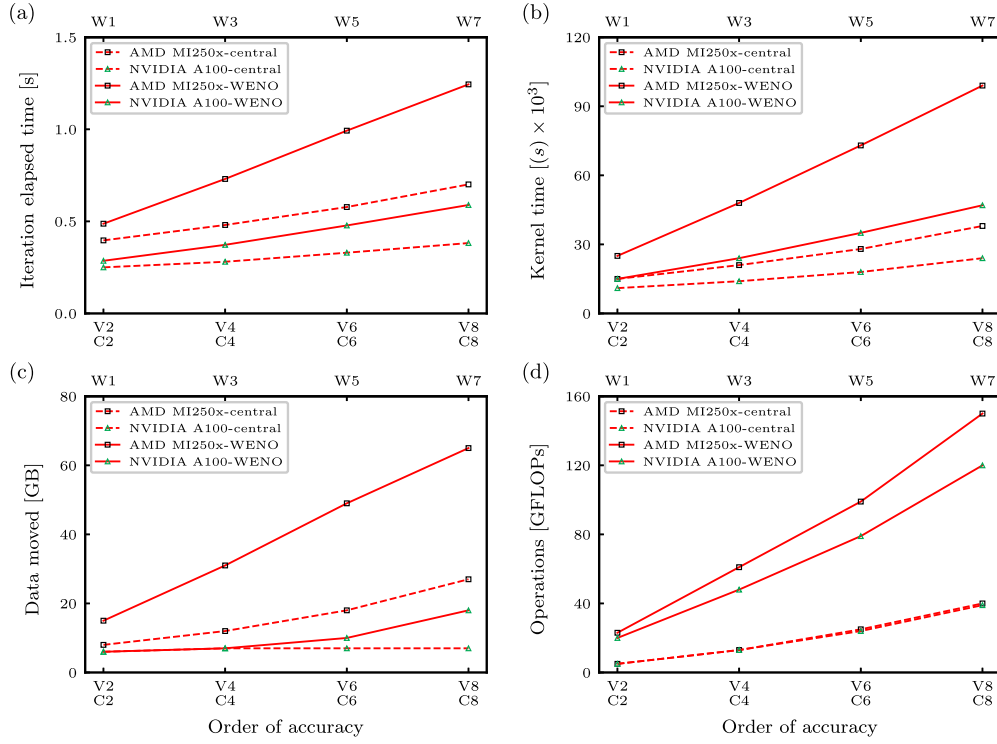
Fig. 9. Effect of order of accuracy for grid: $(420 \times 250 \times 320)$. *(a)* Effect on average time per iteration (s). *(b)* Effect on euler_y processing time $(s \times 10^3)$. *(c)* Effect on the data moved to and from the HBM (GB) for the euler_y kernel. *(d)* Effect on the number of operations (GFLOPs) for the euler_y kernel. All values taken for MI250x GCD and A100 GPU for central and WENO flux computation schemes. x-labels refer to the numerical accuracies of viscous terms $(V2, V4, V6, V8)$ or convective central schemes $(C2, C4, C6, C8)$ and convective WENO schemes $(W1, W3, W5, W7)$. Green points refer to A100 GPU results and black points refer to MI250X GCD results. Solid lines correspond to WENO results and dashed lines correspond to central scheme results.

the MI250x GCD shows more than linear behavior. Considering WENO, the A100 GPU still moves much less data, but the increase with numerical order is now more than linear for the A100 GPU, while a relatively perfect linear behavior is visible for the MI250x GCD. In general, the different sizes of data moved benefit the A100 GPU's execution time greatly, as the peak bandwidth times for the two devices are almost similar. This is typically due to better compiler translation and a more efficient device caching mechanism. Fig. 9d shows the number of operations. The differences between AMD and NVIDIA are much less pronounced: there is almost no difference for the central scheme, while there is a moderate but measurable reduction in the number of operations for the A100 GPU when dealing with WENO schemes. All in all, as discussed for the roofline analysis, the code is mostly memory-bound and the main performance advantage of NVIDIA seems to be the large reduction in the amount of data moved, while from a programming point of view it consists of the same set of operations.

STREAmS-2 production runs often include hybrid schemes, where the central scheme is used in smooth flow regions and WENO scheme is activated close to shocks. Hence, different numerical schemes are used depending on the flow region. The percentage activation of WENO, which is more computationally demanding than the central scheme, depends on the shock sensor threshold and the flow type. Therefore, the percentage of WENO activation is not known a priori and may even vary during computation. In order to understand its impact on the run times, we look at the performance of a kernel (euler_y) and the full solver for different WENO activation percentages using the hybrid scheme.

Fig. 10a shows the average iteration time as a function of the WENO activation percentage. For reference, we also report horizontal lines corresponding to central (dashed lines) and WENO (solid lines) execution times, which are also plotted for each GPU. As expected, the results are close to the central scheme for minimal WENO activations, however we find a non-monotonic trend where the maximum time step is observed when using a hybrid scheme rather than for the full WENO scheme. In particular, hybrid configurations with 40% or more WENO activation result in a computing time larger than full WENO by 20% for A100 GPUs and by 40% for MI250x GCD. This might seem counter-intuitive and requires further explanation, as two different factors are contributing to the increased time step when using WENO. The first is the additional number of operations required by WENO scheme, and second is the thread divergence, namely the fact that threads perform different operations depending if they are calculating a smooth flow region or a shocked region.

Figs. 10b and 10c look separately at the effects of thread divergence for the A100 GPU and MI250x GCD respectively using the euler_y kernel, where the hybrid schemes have a more pronounced impact. The horizontal lines here, similar to 10a, correspond to the central (dashed lines) and WENO (solid lines) execution times, but for the kernel in question. For the hybrid configuration, in addition to the measured time (solid lines), we also plot the estimated time (dashed lines), which should increase linearly with the WENO activation percentage. As previously noted, the crossover is around 40% of the WENO activation for both GPUs, with an increase of the computing time of up to 30% compared to the full WENO case. All in all, both the NVIDIA and AMD GPUs show an effect of thread divergence but the extent seen is lower in the NVIDIA GPUs.

4.2. Scalability

The performance of an efficient DNS code ultimately depends on how well it scales across a multi-node architecture, where each node contains multiple GPUs. We evaluate the inter-node parallel performance of STREAmS-2 based on the CUDA Fortran and HIP backends. Scalability is mea-
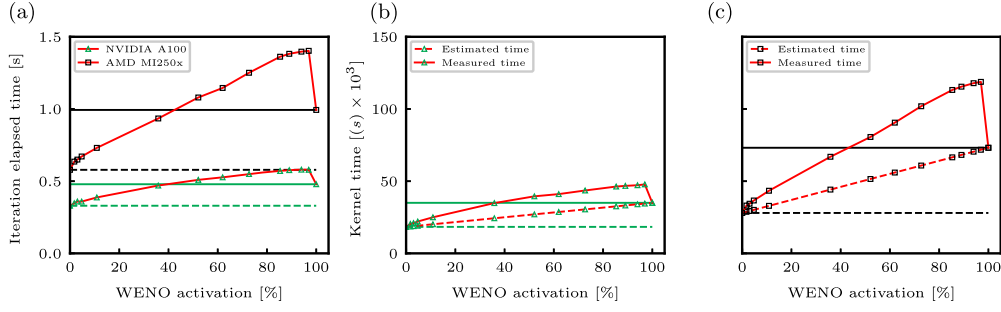
Fig. 10. Effect of WENO activation on thread divergence for grid: $(420 \times 250 \times 320)$. *(a)* Effect on average time per iteration (s). *(b)* Effect on euler_y kernel processing time for NVIDIA A100 GPU $(s \times 10^3)$. *(c)* Effect on euler_y kernel processing time for AMD MI250x GCD $(s \times 10^3)$.
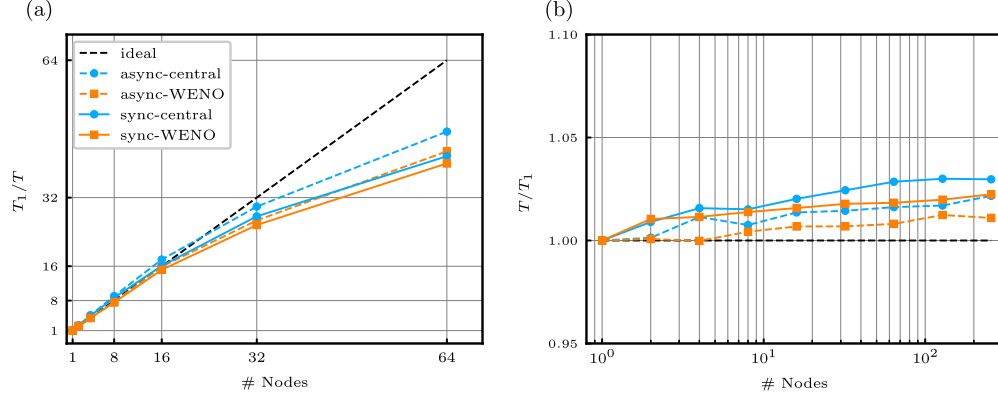

Fig. 11. Strong *(a)* and weak *(b)* scalability plots for STREAmS-2 in synchronous and asynchronous modes for central and WENO flux schemes on the LUMI GPU partition (LUMI-G). The strong scaling is obtained as the ratio of the elapsed time for one node (T_1) to the elapsed time (T) for a grid of about 1.1 billion points. Weak scaling is obtained as the ratio of elapsed time (T) to elapsed time for one node (T_1), with each node holding a grid of about 1.1 billion points. Each node consists of 4 AMD MI250x GPUs or 8 GCDs with a total of 512 GiB of memory.

Table 7
Hardware and software specifications for the scalability runs on LUMI and Leonardo clusters.

	LUMI	Leonardo
GPU	AMD MI250X	NVIDIA A100 SXM4
GPUs/node	4 (8 GCD)	4
Interconnect	HPE Slingshot 11	NVIDIA HDR
Compiler	PrgEnv-gnu/GCC 11.2.0	NVIDIA HPC-SDK 23.1
MPI	Cray MPICH 8.1.18	OpenMPI 4.1.4
CUDA/ROCm	ROCm 5.0.2	CUDA 11.8

sured on EuroHPC JU pre-exascale supercomputers, LUMI (CSC) and Leonardo (Cineca). Table 7 gives an overview of the hardware and software specifications used to perform the scalability runs in this work.

A common bottleneck in multi-node simulations is the MPI communication required to evaluate the derivatives. In this work, we use a GPU-aware communication model to perform these exchanges efficiently, avoiding unnecessary transfers between CPU and GPU. STREAmS-2 can perform this communication in two modes, synchronous and asynchronous, and more details on these two communication patterns are available in the reference publication of STREAmS-1 [17]. We evaluate the performance differences between these two modes. In addition, we present the differences between the central and WENO flux evaluation modes.

Each LUMI node on the LUMI-G partition consists of eight GCDs (four GPUs in total). We compile STREAmS-2 by accessing the HIP library through the HIPFort interface (hipfc wrapper compiler) which is compiled with GCC 11.2.0 and ROCm 5.0.2 (see Table 7). Figs. 11a and 11b illustrate the strong and weak scalability performance of the HIP backend. For the strong scalability we use a grid consisting of 1.1 billion points, which for the reference case of a single node corresponds to a memory utilization of about 91% out of the available 512 GiB (eight GCDs). For weak scalability, we use the same grid for each node.

The strong scalability plot (Fig. 11a) shows an initial superlinearity up to 16 nodes for all cases. Beyond 16 nodes, the profiles begin to diverge from the linear case. Considering the strong scaling efficiency, which is the percentage ratio of the achieved speedup to the ideal speedup, we achieve more than 80% up to 32 nodes. Asynchronous communication has a large advantage over the synchronous case for the central scheme, while it is slightly better for WENO. As for the weak scalability performance (Fig. 11b), we see a significant weak scaling efficiency (percentage ratio of ideal to achieved speedup) greater than 97% in all cases. Asynchronous communication has the upper hand again in both the central and WENO cases as the number of nodes increases, with efficiencies above 98%.

Each Leonardo node on the GPU partition consists of four GPUs. The backend is compiled using the NVIDIA HPC-SDK compiler (see Table 7). Figs. 12a and 12b illustrate the strong and weak scalability performance of the CUDA Fortran backend. We use the same strategy to perform the
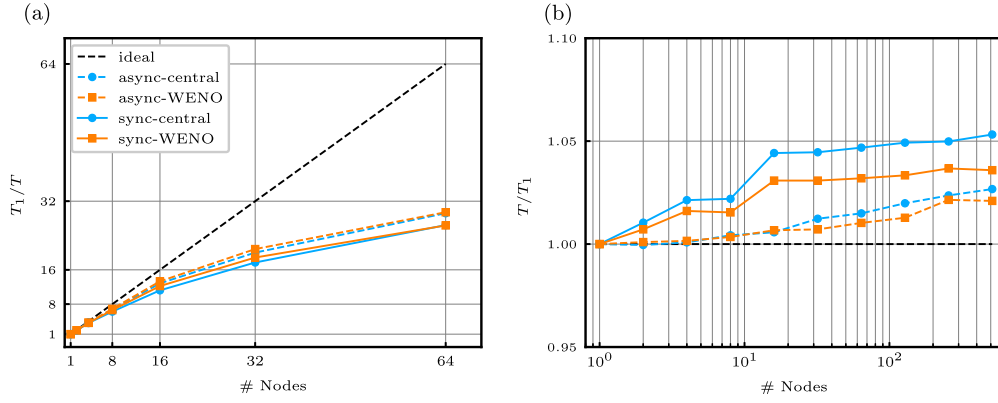
(a)

(b)

Fig. 12. Strong *(a)* and weak *(b)* scalability plots for STREAmS-2 in synchronous and asynchronous modes for central and WENO flux schemes on Leonardo GPU partition. The strong scaling is obtained as the ratio of the elapsed time for one node (T_1) to the elapsed time (T) for a grid of about 0.6 billion points. Weak scaling is obtained as the ratio of elapsed time (T) to elapsed time for one node (T_1), with each node holding a grid of about 0.6 billion points. Each node consists of 4 NVIDIA A100 GPUs with a total of 256 GiB of memory.

scalability evaluation, similar to the LUMI case, but we consider a grid of 0.6 billion points, which for the reference case of a single node corresponds to a memory usage of about 90% of the available 256 GiB (four GPUs).

On the strong scalability plot based on Fig. 12a, we see that the achieved speed deviates from the ideal line at around 16 nodes. The strong scaling efficiencies at 32 nodes drop to around 55-65% for all cases. However, the weak scaling results are excellent, with efficiencies for the asynchronous case greater than 98%. Again, the asynchronous mode performs better for both strong and weak scaling.

Comparing Figs. 11a and 12a we see that STREAmS-2 has better strong scalability performance for the HIP backend on LUMI. In particular, superlinearity only occurs in this case. This result can be attributed to the different behavior of the GPUs when varying the number of grid points processed, as shown in Fig. 7. We can see that as the number of points increases, the grind time for the NVIDIA GPU continues to decrease. For the AMD counterpart, however, it initially decreases, but then increases again. In particular, the optimal grind time for the NVIDIA card occurs when the number of grid points per GPU is at its highest, but for the AMD GPU it occurs when it is relatively lower. When the number of GPUs is increased for strong scaling, the effect of reducing the number of grid points per GPU is superimposed on the MPI communication overhead. For the NVIDIA case, the two effects are always additive, resulting in a progressively decreasing scaling efficiency. For the AMD case, the situation is more complex: in the first part of the scaling, the performance improvement related to the grid reduction seems to outweigh the relative increase in communication overhead, while in the second part, the grid reduction effect and the communication overhead have the same effect, leading to the final performance degradation.

The weak scalability comparison, based on Figs. 11b and 12b, shows impressive efficiencies for both synchronous and asynchronous modes. However, the asynchronous mode always reduces the efficiency loss compared to the synchronous mode. For the CUDA Fortran backend, at the highest number of nodes, this reduction is around 1.4% for WENO and 2.5% for central. Similarly, for the HIP backend, this reduction is less pronounced but still measurable at around 1%. On the other hand, using asynchronous mode has a greater advantage when using smaller grid sizes per GPU, as evidenced by the strong scalability plots for both backends.

The results have important implications for the capabilities of the EuroHPC JU pre-exascale clusters. From Fig. 11b, the number of points based on 256 nodes is about 301 billion. Based on this speedup trend, we can efficiently run cases on the order of 1 trillion points, which will certainly help to study supersonic/hypersonic flow physics on an unprecedented scale.

5. Conclusions

With the aim of taking full advantage of the powerful modern HPC GPU architectures, we present an enhanced version of STREAmS-2 for the study of compressible turbulent wall-bounded flows. The GPUs under scrutiny are based on the NVIDIA and AMD architectures. The framework and strategies outlined in this paper should allow STREAmS-2 to tackle complex flow problems through high fidelity simulations of an unprecedented level, usually considered implausible. We have described a development model built on a multi-equation, multi-backend framework, where the primary code version is based on CUDA Fortran, allowing efficient use of NVIDIA GPUs while maintaining good readability, which is crucial especially when developing new code features. On the other hand, a Python tool was developed to convert the CUDA Fortran code into traditional CPU and HIP paradigms, the latter suitable for AMD GPUs. It was essential to follow an object-based design and strict programming guidelines to ensure a clean conversion process.

The single-GPU evaluations show the impressive performance of GPUs compared to traditional CPUs. In addition, the detailed investigations allow interesting comparisons to be made between the two GPUs currently at the heart of many EuroHPC systems, the NVIDIA A100 and the AMD MI250x. In general, it was found that the gap between peak performance and actual STREAmS-2 performance is generally smaller for NVIDIA cards. Roofline analysis allows us to partially trace the reason for this behavior and shows that, somewhat unexpectedly, the computational intensity of the core kernels is significantly different for the two architectures. Additional single-GPU comparisons are performed to assess the impact of grid size, number of parallelized loops, thread masking and thread divergence. The results obtained can be of interest to users in configuring their execution setup and to developers in efficiently programming new algorithms.

The parallel performance has been evaluated in terms of strong and weak scaling, taking into account synchronous and asynchronous communication patterns and running on the two largest EuroHPC clusters, namely LUMI (CSC) and Leonardo (Cineca). The strong scalability shows more than 80% efficiency up to 16 nodes for Leonardo and up to 32 nodes for LUMI. Weak scalability shows an impressive efficiency of over 95% up to the maximum number of nodes tested (256 for LUMI and 512 for Leonardo). Enabling the asynchronous pattern for communication-computation overlap allows significant improvements. Overall, the final performances on these clusters allow us to understand that impressive computational grids can

be adopted, potentially approaching trillions of grid points. Thus, the physical and engineering problems that will be addressed by STREAmS-2 have the full potential to cover new areas and applications of compressible fluid dynamics.

CRediT authorship contribution statement

Srikanth Sathyanarayana: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft, Writing – review & editing. **Matteo Bernardini:** Funding acquisition, Methodology, Supervision, Writing – review & editing. **Davide Modesti:** Formal analysis, Project administration, Supervision, Writing – review & editing. **Sergio Pirozzoli:** Conceptualization, Resources, Software, Validation. **Francesco Salvadore:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Data availability

The authors are unable or have chosen not to specify which data has been used.

References

[1] TOP500, https://www.top500.org/lists/top500/2022/11/, 2022. (Accessed 25 February 2023).
[2] CUDA, https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2023. (Accessed 25 February 2023).
[3] X. Zhu, E. Phillips, V. Spandan, J. Donners, G. Ruetsch, J. Romero, R. Ostilla-Mónico, Y. Yang, D. Lohse, R. Verzicco, M. Fatica, R. Stevens, AFiD-GPU: a versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters, Comput. Phys. Commun. 229 (2018) 199–210.
[4] P. Costa, E. Phillips, L. Brandt, M. Fatica, GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows, Comput. Math. Appl. (2020).
[5] CUDA Fortran, https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/, 2023. (Accessed 25 February 2023).
[6] OpenACC, https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/, 2023. (Accessed 25 February 2023).
[7] F. Witherden, A. Farrington, P. Vincent, PyFR: an open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach, Comput. Phys. Commun. 185 (11) (2014) 3028–3040.
[8] J. Romero, J. Crabill, J. Watkins, F. Witherden, A. Jameson, ZEFR: a GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method, Comput. Phys. Commun. 250 (2020) 107169.
[9] F. De Vanna, F. Avanzi, M. Cogo, S. Sandrin, M. Bettencourt, F. Picano, E. Benini, URANOS: a GPU accelerated Navier-Stokes solver for compressible wall-bounded flows, Comput. Phys. Commun. (2023) 108717.
[10] E. Otero, J. Gong, M. Min, P. Fischer, P. Schlatter, E. Laure, OpenACC acceleration for the PN–PN-2 algorithm in Nek5000, J. Parallel Distrib. Comput. 132 (2019) 69–78.
[11] N. Jansson, M. Karp, A. Podobas, S. Markidis, P. Schlatter, Neko: a modern, portable, and scalable framework for high-fidelity computational fluid dynamics, arXiv preprint, arXiv:2107.01243, 2021.
[12] HIP, C++ heterogeneous-compute interface for portability, https://github.com/ROCm-Developer-Tools/HIP/, 2023. (Accessed 25 February 2023).
[13] OpenCL, https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf, 2023. (Accessed 25 February 2023).
[14] G.R. Mudalige, I. Reguly, S.P. Jammy, C.T. Jacobs, M.B. Giles, N.D. Sandham, Large-scale performance of a DSL-based multi-block structured-mesh application for direct numerical simulation, J. Parallel Distrib. Comput. 131 (2019) 130–146.
[15] G.A. Bres, S.T. Bose, C.B. Ivey, M. Emory, F. Ham, GPU-accelerated large-eddy simulations of supersonic jets from twin rectangular nozzle, 2022.
[16] F. Salvadore, M. Bernardini, M. Botti, GPU accelerated flow solver for direct numerical simulation of turbulent flows, J. Comput. Phys. 235 (2013) 129–142.
[17] M. Bernardini, D. Modesti, F. Salvadore, S. Pirozzoli, STREAmS: a high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows, Comput. Phys. Commun. 263 (2021) 107906.
[18] M. Bernardini, D. Modesti, F. Salvadore, S. Sathyanarayana, G. Della Posta, S. Pirozzoli, STREAmS-2.0: supersonic turbulent accelerated Navier-Stokes solver version 2.0, Comput. Phys. Commun. (2023) 108644.
[19] S. Pirozzoli, M. Bernardini, Turbulence in supersonic boundary layers at moderate Reynolds number, J. Fluid Mech. 688 (2011) 120–168.
[20] S. Pirozzoli, M. Bernardini, Probing high-Reynolds-number effects in numerical boundary layers, Phys. Fluids 25 (2) (2013) 021704.
[21] M. Cogo, F. Salvadore, F. Picano, M. Bernardini, Direct numerical simulation of supersonic and hypersonic turbulent boundary layers at moderate-high Reynolds numbers and isothermal wall condition, J. Fluid Mech. 945 (2022) A30.
[22] M. Bernardini, G. Della Posta, F. Salvadore, E. Martelli, Unsteadiness characterisation of shock wave/turbulent boundary-layer interaction at moderate Reynolds number, J. Fluid Mech. 954 (2023) A43.
[23] D. Modesti, S. Pirozzoli, Reynolds and Mach number effects in compressible turbulent channel flow, Int. J. Heat Fluid Flow 59 (2016) 33–49.
[24] M. Bernardini, S. Pirozzoli, P. Orlandi, Compressibility effects on roughness-induced boundary layer transition, Int. J. Heat Fluid Flow 35 (2012) 45–51.
[25] M. Bernardini, S. Pirozzoli, P. Orlandi, S. Lele, Parameterization of boundary-layer transition induced by isolated roughness elements, AIAA J. 52 (10) (2014) 2261–2269.
[26] D. Modesti, S. Sathyanarayana, F. Salvadore, M. Bernardini, Direct numerical simulation of supersonic turbulent flows over rough surfaces, J. Fluid Mech. 942 (2022) A44.
[27] S. Pirozzoli, Generalized conservative approximations of split convective derivative operators, J. Comput. Phys. 229 (19) (2010) 7180–7190.
[28] Y. Tamaki, Y. Kuya, S. Kawai, Comprehensive analysis of entropy conservation property of non-dissipative schemes for compressible flows: KEEP scheme redefined, J. Comput. Phys. 468 (2022) 111494.
[29] B. McBride, NASA Glenn Coefficients for Calculating Thermodynamic Properties of Individual Species, National Aeronautics and Space Administration, John H. Glenn Research Center, 2002.
[30] oneAPI, https://www.oneapi.io/, 2023. (Accessed 31 March 2023).
[31] OpenMP, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf/, 2023. (Accessed 25 February 2023).

[32] kokkos, https://github.com/kokkos/kokkos, 2023. (Accessed 31 March 2023).

[33] Legion, https://legion.stanford.edu/, 2023. (Accessed 31 March 2023).

[34] OpenSYCL, https://github.com/OpenSYCL/OpenSYCL, 2023. (Accessed 31 March 2023).

[35] alpaka, https://github.com/alpaka-group/alpaka, 2023. (Accessed 31 March 2023).

[36] RAJA, https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms, 2023. (Accessed 31 March 2023).

[37] HIPIFY: convert CUDA to portable C++ code, https://github.com/ROCm-Developer-Tools/HIPIFY, 2023. (Accessed 25 February 2023).

[38] HIPFort, https://github.com/ROCmSoftwarePlatform/hipfort/, 2023. (Accessed 25 February 2023).

[39] hipCUB, https://github.com/ROCmSoftwarePlatform/hipCUB/, 2023. (Accessed 25 February 2023).

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

[41] F. Salvadore, G. Rossi, S. Sathyanarayana, M. Bernardini, OpenMP offload toward the exascale using Intel® GPU Max 1550: evaluation of STREAmS compressible solver, J. Supercomput. 80 (14) (2024) 21094–21127.

[42] EUROHPC JU, https://eurohpc-ju.europa.eu/about/our-supercomputers_en, 2023. (Accessed 31 March 2023).

[43] C. Yang, T. Kurth, S. Williams, Hierarchical roofline analysis for GPUs: accelerating performance optimization for the NERSC-9 perlmutter system, Concurr. Comput., Pract. Exp. 32 (20) (2020) e5547.

Srikanth Sathyanarayana is pursuing his Ph.D. at the Department of Mechanical and Aerospace Engineering at Sapienza University of Rome. His Ph.D. topic deals with the study of distributed surface roughness in supersonic turbulent flow using Direct Numerical Simulations (DNS). As part of his PhD thesis, he also worked on GPU portable methods to efficiently target the DNS code to NVIDIA and AMD GPU architectures. His research interests generally include High Performance Computing, and discrete adjoint methods for steady and unsteady flows.

Matteo Bernardini is an Associate Professor of Fluid Mechanics at the Department of Mechanical and Aerospace Engineering of the University of Rome "La Sapienza". He received his Ph.D. degree in Theoretical and Applied Mechanics from the same university in 2010. His research interests include DNS/LES of compressible turbulent flows, high-Reynolds number turbulent boundary layers, shock-wave/boundary-layer interactions, computational aeroacoustics, roughness-induced transition, solid rocket motors, and high performance computing.

Davide Modesti is an Assistant Professor in the Faculty of Aerospace Engineering at Delft University of Technology and at the Gran Sasso Science Institute in Italy. He received his PhD in Theoretical and Applied Mechanics from La Sapienza Università di Roma in 2017, where he carried out direct numerical simulations of compressible wall-bounded flows at high Reynolds numbers. After completing his PhD, he has been a Research Fellow at DynFluid laboratory in Paris and at the University of Melbourne, before joining TU Delft in 2020. His research focuses on high-fidelity simulations of incompressible and compressible wall-bounded turbulent flows, and on the development of numerical methods for computational fluid dynamics.

Sergio Pirozzoli is a full professor of Fluid Dynamics at Sapienza University of Rome. His research interests include numerical simulation of turbulent incompressible and compressible flows and high performance computing. During his career, he has given contributions to the numerical study of shock/boundary layer interactions by developing novel energy-consistent and shock-capturing schemes and editing a paper on "Numerical Methods for High-Speed Flows" in the Annual Review of Fluid Mechanics in 2011. In 2016 he was elected Fellow of the American Physical Society-Division of Fluid Dynamics with a mention of his work on compressible flows. He is currently involved in the direct numerical simulation of incompressible wall-bounded flows at high Reynolds number.

Francesco Salvadore is an HPC specialist at Cineca. He is part of the High Performance Computing department and a lecturer of specialized courses. He graduated with honors in Mechanical Engineering from the University of Rome Sapienza. He obtained his PhD in Theoretical and Applied Mechanics at Sapienza in 2007 with a thesis on numerical fluid dynamics of reactive flows. He has extensive experience in algorithms and implementation of methods for computational physics, high performance computing, code optimization, parallel programming paradigms, and heterogeneous computing.