Scalable High-Fidelity Simulation of Turbulence With Neko Using Accelerators

Niclas Jansson
PDC Center for High Performance Computing
KTH Royal Institute of Technology
Stockholm, Sweden
njansson@kth.se

Martin Karp
Computational Science and Technology
KTH Royal Institute of Technology
Stockholm, Sweden
makarp@kth.se

Jacob Wahlgren
Computational Science and Technology
KTH Royal Institute of Technology
Stockholm, Sweden
jacobwah@kth.se

Stefano Markidis
Computational Science and Technology
KTH Royal Institute of Technology
Stockholm, Sweden
markidis@kth.se

Philipp Schlatter
Engineering Mechanics
KTH Royal Institute of Technology
Stockholm, Sweden
pschlatt@mech.kth.se

***Abstract*—Recent trends and advancements in including more diverse and heterogeneous hardware in High-Performance Computing are challenging scientific software developers in their pursuit of efficient numerical methods with sustained performance across a diverse set of platforms. As a result, researchers are today forced to re-factor their codes to leverage these powerful new heterogeneous systems. We present Neko – a portable framework for high-fidelity spectral element flow simulations. Unlike prior works, Neko adopts a modern object-oriented Fortran 2008 approach, allowing multi-tier abstractions of the solver stack and facilitating various hardware backends ranging from general-purpose processors, accelerators down to exotic vector processors and Field-Programmable Gate Arrays (FPGAs). Focusing on the performance and portability of Neko, we describe the framework's device abstraction layer managing device memory, data transfer and kernel launches from Fortran, allowing for a solver written in a hardware-neutral yet performant way. Accelerator specific optimisations are also discussed, with auto-tuning of key kernels and various communication strategies using device-aware MPI. Finally, we present performance measurements on a wide range of computing platforms, including the EuroHPC pre-exascale system LUMI, where Neko achieves excellent parallel efficiency for a large DNS of turbulent fluid flow using up to 80% of the entire LUMI supercomputer.**

I. Introduction

With exascale computing capabilities on the horizon, we have transitioned to more heterogeneous architectures. Traditional homogeneous scalar processing machines are replaced with heterogeneous machines that combine scalar processors with various accelerators, such as GPUs. While offering high theoretical peak performance and high memory bandwidth, to efficiently exploit these systems, well-suited numerical algorithms, complex programming models and significant programming investments are necessary. Furthermore, most known pre and exascale systems currently planned or installed, e.g. Frontier and LUMI, contain a substantial amount of accelerators. Thus, the challenge of porting and tuning scientific codes for these new platforms can no longer be ignored.

Computational Fluid Dynamics (CFD) is a natural driver for exascale computing, with a virtually unbounded need for computational resources for accurate simulation of turbulent fluid flow, both for academic and engineering usage. However, established CFD codes build on years of verification and validation of their underlying numerical methods, potentially preventing a complete rewrite of a code base and rendering disruptive code changes a delicate task. Therefore, porting established codes to accelerators poses several interdisciplinary challenges, from formulating suitable numerical methods, performing hardware-specific tuning to applying sound software engineering practices to cope with disruptive code changes.

In this paper, we describe our work on developing a device abstraction layer for Neko – a portable framework for high-fidelity spectral element flow simulations, enabling scalable high-fidelity turbulence simulations on accelerated systems.

The outline of the paper is the following; In Section II an overview of the current state of the art for porting CFD software to accelerators is presented. A description of Neko is given in Section III, following by a description of the accelerator strategy and implementation in Section IV. A performance evaluation and conclusions outlining future work is given in Section V - VI.

II. Current state of the art

Today, most CFD solvers with the ambition of targeting various heterogeneous platforms in a performant way are trying to address the related challenges by putting the platform abstraction layer on the actual kernel implementation in their solvers, and utilize generic libraries. Prime examples are, Kokkos [1], a library consisting of a performance portable abstraction layer in C++ targeting both CPU and GPU architectures and OCCA [2], a library and DSL, or rather a kernel language, with just-in-time compilation for various backends. Both have successfully been used in various CFD codes, e.g. NekRS [3] which is built on top of OCCA. Using these

libraries enables performance portability, but as a core part in the software design, relying too much on their features might introduce dependencies deep in the software stack.

Accommodating an external abstraction layer in an application requires non-trivial integration at a deep level, either via specific data types, e.g. Kokkos-arrays or via DSL snippets for the OCCA library. Abstraction requiring specific data types will implicitly enforce consumer applications to use the same programming language or end up with a mixed-language codebase that might become a maintenance nightmare. With a DSL approach, the impact is less severe and can be limited only to affect specific computational kernels. However, for all these approaches, there is always a question about portability and sustainability.

The success and longevity of several established CFD codes can in part be attributed to the design choice of using a standardised programming language, e.g., Fortran 77. Thus, as long as any future platform has a standard-conforming Fortran 77 compiler, we would also have a working CFD solver on that platform. Hence, portability and long-term sustainability could be the unfortunate Achilles' heel of abstraction layers such as Kokkos and OCCA. Furthermore, it is also a question about programming models and their support on current and future platforms. Today, most heterogeneous systems with accelerators are programmed using a model that offloads computation from the host (CPU) to the accelerator (often a GPU). To benefit from these systems, applications must consider how the offloaded kernels are computed, and as necessary, how to optimise memory transfers between host and accelerator. What will happen if we suddenly get a new accelerator, which does not operate following the current offloading convention?

Given the challenges of finding a portable programming abstraction, we argue that another option is to design application-specific abstraction layers to facilitate the ease of use of different hardware-specific kernels and solvers for a particular programming model.

III. Neko

To address these challenges and to enable high-fidelity fluid simulations on accelerated systems like Frontier and LUMI, we have developed Neko, a portable framework for high-order spectral element-based simulations. The framework is implemented in modern Fortran 2008 and adopts a modern object-oriented approach allowing for multi-tier abstractions of the solver stack and facilitating various hardware backends [4]–[7]. Using Fortran as the language of choice instead of recently more popular languages such as C++ or Python might at first seem like an odd choice, particularly for developing a new code. However, Neko has its roots in the spectral element code Nek5000 [8] from UChicago/ANL introduced in the mid-nineties, tracing its origins to MIT's older code NEKTON 2.0. Furthermore, research groups at KTH have extensively used the scalable Nek5000 and also further developed its Fortran 77 codebase, thus leaving a non-negligible trace of more than thirty years of verified and validated Fortran code, which, if rewritten into, e.g., C++ would have to go through a very expensive and time-consuming revalidation and reverification process before it could be used in production. Therefore, by using modern Fortran, already validated Fortran 77 kernels can directly be integrated into Neko, with only a minimal revalidation process.

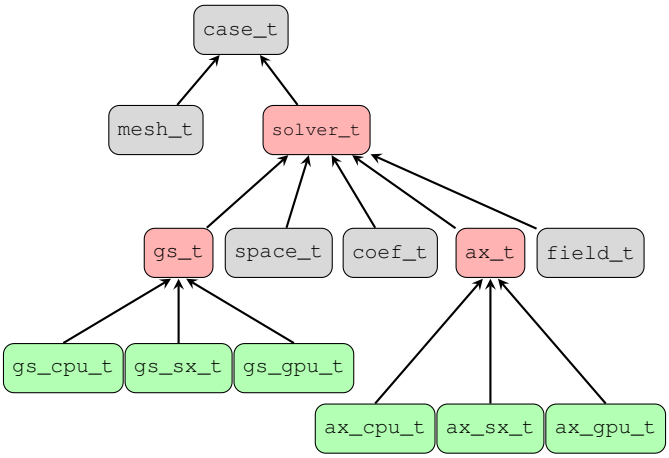


Fig. 1: An illustration of a canonical flow case in Neko, with typical derived types (gray), abstract types (red) with actual implementation in extended derived types (green).

Neko integrates in time the incompressible Navier–Stokes equations, ensuring single-core/accelerator efficiency via fast tensor product operator evaluations. For high-order methods, assembling either the local element matrix or the full stiffness matrix is prohibitively expensive. Therefore, a key to achieving good performance in spectral element methods is to consider a matrix-free formulation, where one always works with the unassembled matrix on a per-element basis. Gather–scatter operations ensure the continuity of functions on the element level, operating on both intra-node and inter-node element data.

A. Abstraction layer

When designing a flexible and maintainable framework for computational science, a major issue is finding the right level of abstraction. Too many levels might degrade performance, while too few results in a code base with many specialised kernels at a high maintenance cost. The weak form of the equation used in the Spectral Element Method allows Neko to recast equations in the form of the abstract problem to keep the abstractions at the top level and reduce the amount of platform-dependent kernels to a minimum. In Neko, this is realised using abstract Fortran types, with deferred implementations of required procedures. For example, to allow for different formulations of a simulation's governing equations, Neko provides an abstract type, defining the abstract problem's matrix-vector product. The type comes with a deferred procedure `compute` that would return the action of multiplying the stiffness matrix of a given equation with a vector. In a typical object-oriented fashion, whenever a routine needs a matrix-vector product, it is always expressed as a call to `compute` on the abstract

base type and never on the actual concrete implementation. Abstract types are all defined at the top level in the solver stack (as illustrated in Fig. 1) during initialisation and represent large, compute-intensive kernels, thus reducing overhead costs associated with the abstraction layer. Furthermore, this abstraction also accommodates the possibility of providing tuned matrix-vector products for specific hardware, only providing a particular implementation of `compute` without having to modify the entire solver stack, as illustrated in Fig. 1 with the hardware-specific green boxes.

IV. Implementation on accelerators

Regardless of the abstraction in Neko, modern Fortran will not provide a method for interfacing with accelerators; thus, a second programming model is needed. Despite a popular choice when porting Fortran codes to accelerators, a decision was taken early on in the development of Neko not to rely on vendor-specific solutions (e.g. CUDA Fortran) for the accelerator implementation due to their reduced portability. Another popular strategy to enable accelerators in Fortran codes is to use directives-based approaches, e.g. OpenACC or OpenMP, to offload work to the accelerator. Albeit promising result has been shown for porting Nek5000 to GPUs [9], initial testing with Neko revealed reduced performance and compiler immaturity, not seldom due to our extensive use of modern Fortran's object-oriented features.

A. Device abstraction layer

Instead of vendor-specific solutions or directives-based approaches, Neko uses a device abstraction layer to manage device memory, data transfer and kernel launches from Fortran.

Designed based on the necessary memory management features needed on the Fortran side; functionality to allocate/deallocate device memory, associate an array in Fortran with an array allocated on the device and methods for moving data between memory spaces. Internally the interface uses pointers (`c_ptr`) from Fortran's c-bindings to keep track of device memory, and behind this interface, Neko calls the native accelerator implementation written in, e.g. CUDA, HIP or OpenCL via c-interfaces for each supported accelerator. Interfaces are defined inside Neko for each supported backend, covering all necessary API calls, e.g. memory management and device synchronisation, as well as enumerators defining, e.g. error codes and direction of data transfer as illustrated in Listing 1 for Neko's HIP interface.

This way, we can keep the Fortran side of Neko's device layer close to the native accelerator implementation. For example, to allocate `s` bytes of data on a device and keep track of the allocation with the pointer `ptr`, the interface provides a function `device_alloc(ptr, s)`, which calls the native memory allocation routine depending on which accelerator backend Neko has been configured to support, as illustrated in Listing 2. However, since it is rather uncommon to allocate memory in terms of bytes in Fortran, a couple of helper functions, `device_map(x, x_d, n)`, are defined to map a Fortran array `x` to a device pointer `x_d`. These functions determine the data type of `x`, the size in bytes, and call `device_alloc`. Furthermore, the association of arrays uses a hash table, mapping the location (using `c_loc`) of a Fortran array to the device pointer. Once associated, the device pointer of any array can be retrieved anywhere inside the code base via a simple lookup in the hash table.

Listing 1: Part of Neko's HIP interface.

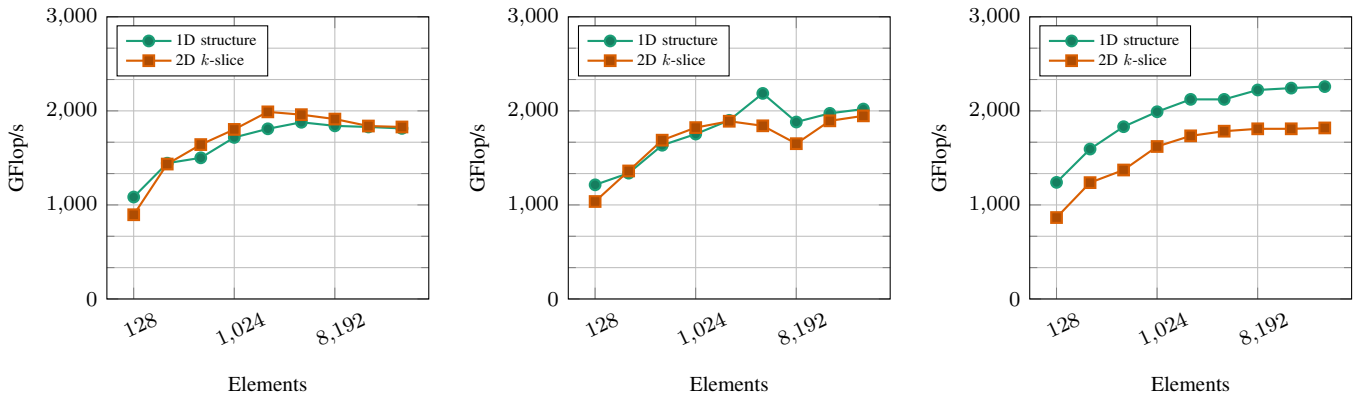
```
!> Enum @a hipError_t
enum, bind(c)
   enumerator :: hipSuccess = 0
   ...
end enum

!> Enum @a hipMemcpyKind
enum, bind(c)
   enumerator :: hipMemcpyHostToHost = 0
   enumerator :: hipMemcpyHostToDevice = 1
   ...
end enum

interface
   integer (c_int) function hipMalloc(ptr_d, s) &
        bind(c, name='hipMalloc')
     use, intrinsic :: iso_c_binding
     implicit none
     type(c_ptr) :: ptr_d
     integer(c_size_t), value :: s
   end function hipMalloc
end interface
```

Listing 2: Memory allocation in the device abstraction layer.

```
!> Allocate memory on the device
subroutine device_alloc(x_d, s)
  type(c_ptr), intent(inout) :: x_d
  integer(c_size_t) :: s
  integer :: ierr
#ifdef HAVE_HIP
  if (hipMalloc(x_d, s) .ne. hipSuccess) then
     call neko_error('Memory allocation on device failed')
  end if
#elif HAVE_CUDA
  if (cudamalloc(x_d, s) .ne. cudaSuccess) then
     call neko_error('Memory allocation on device failed')
  end if
#elif HAVE_OPENCL
  x_d = clCreateBuffer(glb_ctx, CL_MEM_READ_WRITE, &
                       s, C_NULL_PTR, ierr)
  if (ierr .ne. CL_SUCCESS) then
     call neko_error('Memory allocation on device failed')
  end if
#endif
end subroutine device_alloc
```

B. Derived Types

To adhere to Neko's abstraction (see Sect. III-A), most derived types defining a current case to be solved (e.g. fields, coefficient, mesh) need to either be replicated with accelerator versions or to be extended to accommodate for the possibility of keeping track of memory on a device. To avoid replicating most derived types to accommodate for device memory, Neko follows a similar approach as taken in FUN3D [10], whereas the derived type is extended with pointers (`type(c_ptr)`) to keep track of device memory. However, unlike FUN3D, Neko does not create interoperable mirror types (in Fortran and C) to hold information on the device; instead, the device pointer is added directly to the derived type, as illustrated for a field type in Listing 3, with a default value of `NULL` for the device pointer unless device memory is needed (e.g. when executing



(a) 7th order polynomials. **(b) 9th order polynomials.** **(c) 11th order polynomials.**

Fig. 2: Performance in GFlop/s for computing the Poisson operator on a single AMD MI250X GCD, comparing two different shapes of the thread blocks, a one-dimensional structure with a manual calculation of indices or two-dimensional slices along the k direction of a hexahedral element for 7th (a), 9th (b) and 11th (c) order polynomials on different number of elements.

Listing 3: A derived type suitable for both host and device.

```
type field_t
   real(kind=rp), allocatable :: x(:,:,:,:) !< Field data
   type(space_t), pointer :: Xh     !< Function space
   type(mesh_t), pointer :: msh     !< Mesh
   type(dofmap_t), pointer :: dof   !< Dofmap
   type(c_ptr) :: x_d = C_NULL_PTR !< Device pointer
end type field_t
```

on CPUs). This way of using the same derived type for both CPU and GPU executions also reduces the number of almost identical, replicated types in the code base and allows us to keep the hardware-specific code paths at lower levels in the solver abstraction (following the design approach described in Sect. III-A).

C. Kernel Launches and Optimisations

Device kernels are launched from Fortran via c-interfaces, calling a small wrapper function in C or C++ (extern "C" declared), which in turn launches the device kernel. Each routine in Neko with the possibility of executing on an accelerator has a corresponding Fortran procedure for calling the wrapper function, containing the kernel launch. Since each accelerator programming model differs slightly, and to allow for vendor-specific implementations, Neko provides a wrapper function for each supported accelerator backend (currently HIP, CUDA and OpenCL), selected by pre-processor macros in a similar way as illustrated for the device memory allocation in Listing 2. Furthermore, kernel launches are either templated (CUDA/HIP) or rely on pre-processor macros (OpenCL) selecting tuned kernels for runtime-dependent parameters, e.g. the polynomial order of a simulation.

Several important kernels in Neko follow a similar tensor-product structure to the kernels by Świrydowicz et al. [11], and as observed in their work the shape of thread blocks, and how they map to elements can significantly affect the performance of a kernel. On Nvidia hardware, an overall good performance is obtained using the two-dimensional thread block structure from [11], moving the slice along the k-direction of each hexahedral element. On AMD hardware however, performance was less predictable, with the 2D k-slice approach sometimes showing a slow down compared to a simple one-dimensional thread structure with manual calculation of indices in a thread block. Fig. 2 illustrated this performance variation for computing the Poisson operator for varying polynomial orders and problem sizes on one AMD MI250X Graphics Compute Die (GCD).

The results for the Poisson operator show that the optimal launch configuration depends on the polynomial order and the problem size on AMD Hardware. However, in this case, templates nor pre-processor macros will not help in selecting an optimal launch configuration given a polynomial order and problem sizes for a given kernel. Instead, Neko uses auto-tuning in all device kernels. The first time a kernel is invoked for a given polynomial order, several experiments are executed for various launch configurations. Once the fastest configuration is found, it is recorded in the auto-tuner for consecutive kernel launches. Furthermore, the auto-tuner can record the fastest strategy for each polynomial order used in a simulation, supporting different strategies in, e.g. multilevel methods, with varying order depending on the level.

D. Gather-Scatter and Device-Aware MPI

As described in Section III, for an efficient spectral element implementation, all operations are performed in a matrix-free fashion in Neko. Functions are defined on an element level, with replicated data for shared degrees of freedom (dof) on element boundaries, and gather-scatter operations are used to ensure continuity of functions across elements.

Since gather-scatter operations must be performed both for internal elements, local to a node as well as external elements, between nodes, Neko uses an overlapped gather-scatter formu-

Algorithm 1 Overlapped gather-scatter algorithm.

$S \leftarrow \emptyset$
for $i = 1, 2, \ldots, m$ **do**
    Post non-blocking receive on $buf(i)$
    $S \leftarrow S \cup i$
end for
$v \leftarrow gather(shared_{dof}, u_L)$
Post non-blocking sends of v
$v \leftarrow gather(local_{dof}, u_L)$
$w_L \leftarrow scatter(local_{dof}, v)$
while $S \neq \emptyset$ **do**
    for all $j \in S$ **do**
        if non-blocking receive j has completed **then**
            $v \leftarrow gather(shared_{dof}, buf(j))$
            $S \leftarrow S \setminus j$
        end if
    end for
end while
$w_L \leftarrow scatter(shared_{dof}, v)$

lation as given in Algorithm 1. First, all non-blocking receives (`MPI_Irecv`) for shared dofs are posted. The shared dofs are gathered into a buffer v, packed into a set of send buffers and transmitted to neighbours sharing the same dofs using non-blocking send (`MPI_Isend`) operations. During the non-blocking communication, the local dofs are gathered from u_L, the replicated element-wise representation of a function u, into a buffer v, the same buffer as for the shared dofs. The buffer now contains all contributions from both shared and local elements (since both gather operations have been performed) for non-shared dofs, and can be scattered into the corresponding places in the output vector w_L (for the local elements). Once the local operation has been completed, a loop poll each of the posted non-blocking receives until all have been completed. As data is received in the loop, it is directly gathered into the buffer v. Finally, with all outstanding receives completed, the shared dofs are scattered back into the output vector w_L.

Furthermore, gather-scatter operations use a set of local-to-global (lg) and global-to-local (gl) map functions for mapping local indices to global dofs (and the reverse). Thus when performing the *gather* and *scatter* operations in Algorithm 1, a naive implementation e.g. $v(lg(i)) = v(lg(i)) + u(gl(i))$, would result in several indirect accesses via the map functions, thus preventing vectorisation and resulting in poor performance on both processors and accelerators. Neko, therefore, classifies dofs as injective or non-injective, depending on the mesh topology. Non-injective dofs are points located in corners and edges (in three dimensions) and could have an arbitrary number of neighbours to consider while performing the gather-scatter operation. Injective dofs are the points on the interior of an edge or face (in three dimensions) with only a single neighbour to operate on, as illustrated in Fig. 3. The non-injective dofs are stored in variable-length blocks (for each independent dof), and the injective part is stored as a contiguous block of sorted tuples, allowing for efficient use of wide SIMD units or vector registers [5].

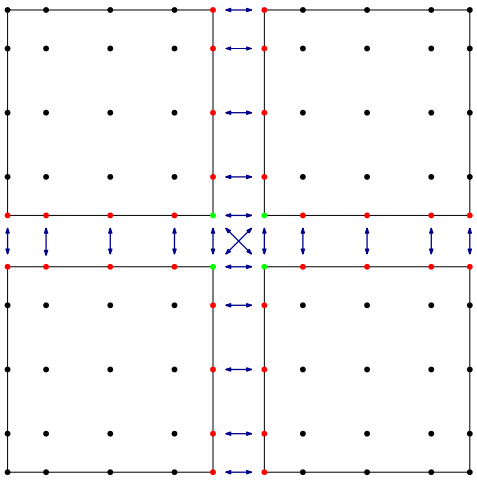


Fig. 3: An illustration of the different types of dofs, edge and corner, on a two dimensional mesh with four 4th order elements, each with five dofs in each direction.

Listing 4: Part of the HIP kernel for gather.

```
const int idx = blockIdx.x * blockDim.x + threadIdx.x;
const int str = blockDim.x * gridDim.x;

for (int i = idx; i < nb; i += str) {
  const int blk_len = b[i];
  const int k = blk_offset[i];
  T tmp = u[gl[k]];
  for (int j = 1; j < blk_len; j++) {
    tmp += u[gl[k + j]];
  }
  v[lg[k]] = tmp;
}

if (shared) {
  for (int i = (offset + idx); i < m ; i += str) {
    v[lg[i]] = u[gl[i]];
  }
}
else {
  if ((idx%2 == 0)) {
    for (int i = (offset + idx); i < m ; i += str) {
      T tmp = u[gl[i]] + u[gl[i+1]];
      v[lg[i]] = tmp;
    }
  }
}
```

Additionally, since each block of dofs contains an independent set of work, this can also be efficiently implemented on accelerators without the need for atomic operations. In Listing 4, we show parts of the HIP kernel for the gather operation. The first loop is for the non-injective blocks, while the second is for the large contiguous injective block. All blocks are stored in the same contiguous array, with the variable `offset` to indicate where the injective data begins.

On systems supporting device-aware MPI, i.e. supporting MPI operations using device memory directly as communication buffers, pack and unpack operations in Algorithm 1 are implemented as GPU kernels allowing for data to be

fully GPU-resident throughout a simulation [12]. Furthermore, Neko's gather-scatter kernel implements four different communication strategies on systems supporting device-aware MPI. Each pack and unpack operation has been designed to support two modes of operation, working on all data to be sent or received or only operating on parts of the data related to one neighbouring rank. This allows Neko to launch the pack and unpack operations concurrently. For example, as soon as a message arrives in the loop at line 11 in Algorithm 1, an unpack kernel could be launched in a device stream to process the received data. However, depending on the network and job distribution, kernel launch latencies might be too costly for launching many smaller unpack kernels as compared to waiting for all communication to finish before unpacking all data. Neko, therefore, uses a runtime selection of the communication strategy. When a gather-scatter object is created, it tests the performance of using fully synchronised pack/unpack operations, asynchronous pack and synchronised unpack operations, synchronised pack and asynchronous unpack operations and finally, fully asynchronous pack and unpack operations. Each MPI rank can select its fastest strategy independently of the other ranks in a simulation in an attempt to address the adverse effects of unbalanced job distributions in the machine.

V. Performance Evaluation

We assess Neko's scalability and performance portability by conducting a strong scalability study on a wide range of computing platforms, ranging from smaller CPU-based systems to the accelerated EuroHPC pre-exascale system LUMI.

A. Performance Portability

To demonstrate the performance portability of Neko's abstract design, we solve the Taylor-Green Vortex problem (TGV) with a Reynolds number equal to 5,000 and measure the average time per time-step once the flow has transitioned to fully turbulent, illustrated in Fig. 4. Using the same Fortran file describing the flow case, we evaluated strong scaling characteristics on a wide range of systems, Piz Daint a Cray XC50 equipped with Nvidia P100 GPUs at CSCS, Beskow a Crax XC40 with Haswell CPUs at PDC, Vulcan a cluster using the SX-Aurora TSUBASA at HLRS, Dardel a Cray EX system with AMD EPYC CPUs at PDC, Alvis a system at C3SE with A100 GPUs, JUWELS a BullSequana XH2000 equipped with Nvidia A100 GPUs at FZJ and LUMI a Cray EX with AMD MI250X GPUs at CSC. The results in Fig. 5 show that Neko demonstrates good strong scaling characteristics and performance portability across a wide range of architectures tested, from various scalar processors, vector processors and accelerators, all driven by the same Fortran code base, but with the platform-specific implementation of key kernels in the leaf routines of the object-oriented solver stack.

Furthermore, the studied Taylor-Green Vortex case used a mesh of 262k spectral elements, thus the CPU results in Fig. 5 demonstrates sustained scalability of Neko with less than 10 elements per CPU core or 10,000 elements per GPU/GCD.

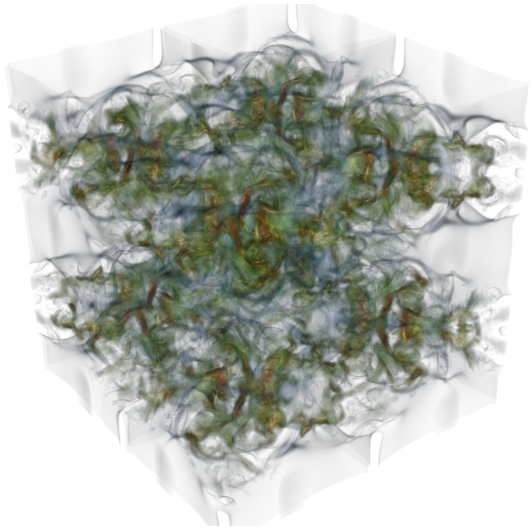


Fig. 4: Volume rendering of the velocity magnitude of the turbulent flow in the Taylor-Green vortex at $Re = 5000$.

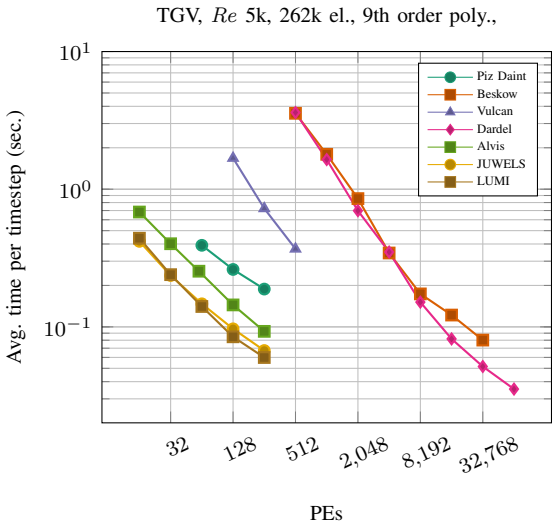


Fig. 5: Strong scaling of Neko for solving the Taylor-Green Vortex problem on a wide range of systems. On the x-axis we show the number of processing elements (PEs) corresponding to one core or one GPU/GCD depending on the platform.

B. Extreme-scale scalability

Extreme-scale strong scalability has been demonstrated on the accelerator partition of the 309 PFlop/s European pre-exascale machine LUMI at CSC. The results in Fig. 6 show that Neko achieves close to 80% parallel efficiency for a large Direct Numerical Simulation (DNS), going from 4096 up to 16,384 AMD MI250X Graphics Compute Dies (GCDs), representing 20%, 40% and 80% of the entire LUMI supercomputer. The simulation studied the flow past a stationary circular cylinder at a Reynolds number of 50,000, one of the highest Reynolds number studied for the particular case with DNS. Furthermore, the simulation were performed using

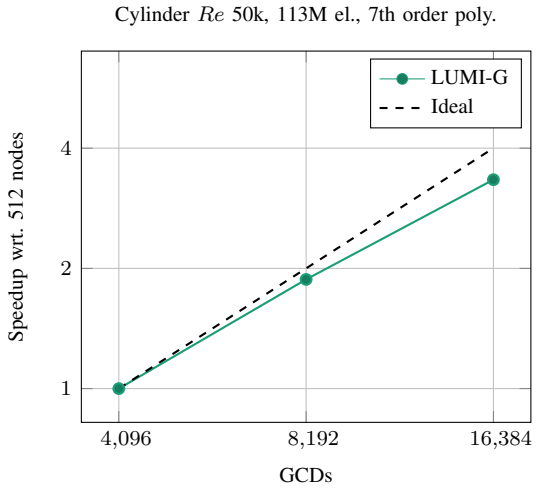


Fig. 6: Strong scaling of Neko for a large DNS simulation of the flow past a circular cylinder using a large fraction of the available AMD MI250X GPUs in the accelerated partition of LUMI (LUMI-G).

a mesh with 113 million spectral elements, using 7th order polynomials, which turns into more than 38 billion unique degrees of freedom, making this one of the largest spectral element simulations performed compared to recently published related work from the NekRS community [13].

VI. Conclusions

In this paper, we have introduced the device abstraction layer in Neko, a modern, high-performance and flexible framework for spectral element based fluid dynamics. The implementation and optimisation strategies necessary for enabling scalable high-fidelity turbulence simulations on accelerators have been discussed, and a performance evaluation on a wide range of computing platforms has been presented, including the EuroHPC pre-exascale system LUMI, where Neko achieves excellent parallel efficiency for a large DNS of turbulent fluid flow using up to 80% of the entire LUMI supercomputer.

Overall, we see that high-order methods and matrix-free approaches are well suited to leverage current powerful accelerator-based systems. However, there are many paths forward for accelerated spectral element methods, and we intend to carefully evaluate several of these with Neko, which offers us the performance, scalability and flexibility to easily develop and evaluated high-fidelity spectral element methods on new accelerated architectures.

Acknowledgment

This work was supported by the European High Performance Computing Joint Undertaking (JU) and Sweden, Denmark, Greece, Germany, Spain under grant reference 101093393 “CEEC – Center of Excellence in Exascale CFD” and the Swedish Research Council under grant reference 2019-04723 “Efficient Algorithms for Exascale Computational Fluid Dynamics”. Financial support from the Swedish e-Science Research Centre Exascale Simulation Software Initiative (SESSI) is also gratefully acknowledged. The experiments were performed on resources provided by Höchstleistungsrechenzentrum Stuttgart (HLRS) and PRACE Research Infrastructure resources Piz Daint hosted by CSCS (Switzerland) and JUWELS hosted by FZJ (Germany) and National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at PDC Center for High Performance Computing and Chalmers Centre for Computational Science and Engineering (C3SE), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. We also acknowledge NAISS and SNIC for awarding this project access to the LUMI supercomputer, owned by the EuroHPC-JU, hosted by CSC (Finland) and the LUMI consortium through a LUMI Sweden XLarge call.

References

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[2] D. S. Medina, A. St-Cyr, and T. Warburton, “OCCA: A unified approach to multi-threading languages,” *arXiv preprint arXiv:1403.0968*, 2014.

[3] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, and T. Warburton, “NekRS, a GPU-accelerated spectral element Navier–Stokes solver,” *Parallel Computing*, vol. 114, p. 102982, 2022.

[4] N. Jansson, M. Karp, A. Podobas, S. Markidis, and P. Schlatter, “Neko: A modern, portable, and scalable framework for high-fidelity computational fluid dynamics,” *arXiv preprint arXiv:2107.01243*, 2021.

[5] N. Jansson, “Spectral Element Simulations on the NEC SX-Aurora TSUBASA,” in *The International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2021. Virtual Event, Republic of Korea: ACM, 2021, pp. 32–39.

[6] M. Karp, A. Podobas, T. Kenter, N. Jansson, C. Plessl, P. Schlatter, and S. Markidis, “A high-fidelity flow solver for unstructured meshes on field-programmable gate arrays: Design, evaluation, and future challenges,” in *International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2022. Virtual Event, Japan: ACM, 2022, p. 125–136.

[7] M. Karp, D. Massaro, N. Jansson, A. Hart, J. Wahlgren, P. Schlatter, and S. Markidis, “Large-scale direct numerical simulations of turbulence using GPUs and modern Fortran,” *The International Journal of High Performance Computing Applications*, 2023 (Online first).

[8] J. W. L. Paul F. Fischer and S. G. Kerkemeier, “nek5000 Web page,” 2008, http://nek5000.mcs.anl.gov.

[9] J. Vincent, J. Gong, M. Karp, A. Peplinski, N. Jansson, A. Podobas, A. Jocksch, J. Yao, F. Hussain, S. Markidis, M. Karlsson, D. Pleiter, E. Laure, and P. Schlatter, “Strong scaling of openacc enabled nek5000 on several gpu based hpc systems,” in *International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia2022. Virtual Event, Japan: ACM, 2022, p. 94–102.

[10] G. Nastac, A. Walden, E. J. Nielsen, and K. Frendi, “Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs,” in *AIAA Scitech 2021 Forum*, 2021.

[11] K. Świrydowicz, N. Chalmers, A. Karakus, and T. Warburton, “Acceleration of tensor-product operations for high-order finite element methods,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 735–757, 2019.

[12] J. Wahlgren, “Using GPU-aware message passing to accelerate high-fidelity fluid simulations,” Master’s thesis, KTH Royal Institute of Technology, 2022.

[13] M. Min, Y.-H. Lan, P. Fischer, E. Merzari, S. Kerkemeier, M. Phillips, T. Rathnayake, A. Novak, D. Gaston, N. Chalmers, and T. Warburton, “Optimization of Full-Core Reactor Simulations on Summit,” ser. SC ’22. IEEE Press, 2022.