https://doi.org/10.5194/egusphere-2023-1079
Preprint. Discussion started: 31 July 2023

Parflow 3.9: development of lightweight embedded DSLs for geoscientific models.

Zbigniew P. Piotrowski[FZJ], Jaro Hokkanen[CSC], Daniel Caviedes-Voullieme[FZJ], Olaf Stein[FZJ], and Stefan Kollet[FZJ]

[FZJ]Forschungszentrum Jülich, Wilhelm-Johnen-Straße, 52428 Jülich, Germany
[CSC]CSC – IT Center for Science Ltd, Keilaranta 14, 02150 Espoo, Finland

Correspondence: Zbigniew Piotrowski (z.piotrowski@fz-juelich.de)

Abstract. Recognizing the leap in high-performance computing with accelerated co-processors, we propose a lightweight approach to adapt legacy codes to next generation hardware and achieve efficiently a high degree of performance portability. We focus on abstracting the computing kernels at the loop levels based on the lightweight, preprocessor-based embedded Domain Specific Language (eDSL) concept in conjunction with Unified Memory management. We outline a set of code pre-adaptations that facilitate the proposed abstraction. In two geophysical code applications programmed in C and Fortran, we demonstrate the efficiency of the eDSL approach in adaptation to NVIDIA GPUs with: native CUDA and Kokkos eDSL backends achieving up to $10-30$ fold speedup. Our experience suggests that the proposed lightweight eDSL code adaptation is less expensive in terms of Full Time Equivalent of effort than adaptation based on complex DSL approaches, even if no earlier GPU competence exists.

1 Introduction

For more than last four decades, the adaptations of scientific codes to new supercomputing architectures have been an ongoing and increasingly complex challenge. The particular scenarios of adaptation varied as dictated by the actual major hardware advancements. For example, the introduction of vector processors enforced the exposition of relatively long loops with arithmetic rather than logical branching. Memory constrains enforced thoughtful coding, supported with (just emerging) software development environments of post-punchcard era. The advent of parallel supercomputers drove the development of multiple program, multiple data and multi-task programming paradigms, along with communication strategies (e.g. SHMEM/MPI) and parallel debuggers. Novel classes of numerical algorithms, especially those capable of utilizing distributed memory and computing chips, gained popularity (Bauer et al., 2021). The parallel supercomputers were soon enhanced with multicore processors, fueling the transition to hybrid (distributed-shared) memory programming models. Interestingly, as the advent of personal computing in the 1980s brought a boost to the software engineering techniques in general, the development of specialized processors for gaming (GPUs) in the 2000s, was seminal to the strategy of offloading computationally-intensive parts of scientific codes to powerful accelerators. This change was accompanied by the alternative, extremely successful IBM Bluegene concept. The latter employed huge numbers of low-power processors equipped with extremely complex and fast interconnects, changing the ratio of time spent in communication and parallel computing. Recent years added to this scenario the tensor hardware and algorithms specialized for exploiting artificial intelligence and machine learning concepts.

The growing supercomputing hardware capabilities enable the development of increasingly complex geoscientific models. The modellers aim to refine the resolution and broaden the range of geophysical scales, and incorporate more biogeophysical and -chemical processes, which also contribute to the computational cost. From the perspective of an implementation, basic representation of mathematical operators or particular geophysical phenomena (constituting computing kernels or a sequence of kernels) evolves rather steadily. However, particular strategies of their optimal representation in memory and execution often vary depending on the heterogeneous architecture under consideration.

Computational design and performance receives diverse attention, depending on the particular area of research and application. Either evolutionary or disruptive code rewrites appear to be preferred, depending on the personal preferences and technical requirements. [1] Notably, extreme (in computational terms) scale applications of weather and climate are subject to extensive software engineering efforts to saturate hardware limits (Fuhrer et al. (2018)). These efforts tend to optimize equipment and total cost of operations (TCO), including in particular energy costs. Optimization of the latter may justify funding of dedicated software engineering, which is an outstanding example of "offsetting" CO2 emissions. However, the majority of existing mid- and small-scale scientific applications can afford neither large software engineering teams, nor abrupt programming paradigm shifts. Instead, they continue to rely on generic solutions provided by hardware vendors. Moreover, even for large and/or operational applications, adapting to disruptive changes (such as new hardware paradigms) is a technical, scientific and management challenge. Fortunately, even at the somewhat niche market of scientific supercomputing, the generic programming models are being steadily and considerably improved. Moreover, the use of relatively simple programming techniques seems to improve productivity of domain scientists, without significant investment in the change of the working environment and corresponding programming skills, as proposed in this work.

Every single scientific application represents an area on a *technology map* spanned by a range of programming languages and hardware classes. Many applications rely on Fortran, the most widespread programming language in Earth system modelling, others utilize the rapidly developing Julia language that embraces modern data processing techniques, e.g. (Sridhar et al. (2021)). Educational efforts, as well as *frontend* interface and postprocessing are often realized by Python implementations, and last but not least many projects use general-purpose C/C++ formulations. On the hardware side, the most common CPUs are increasingly often accompanied by GPU accelerators and AI-focused tensor processors. Some further specialized hardware includes Field Programmable Gate Arrays (FPGAs), ARM or even custom-tailored optical and silicon processors. The technical landscape is thus more complex than ever before and golden technical standards of the past seem to fade away. Consequently, the changing hardware technologies and software solutions to harness call for low-cost adaptation strategies. Herein, we discuss the concept of a simple abstraction layer within an arbitrary programming language, constituting the application agnostic eDSL

[1]For example, a legacy implementation of the Fortran 77-2008 EULAG model (Prusa et al., 2008) has constantly evolved over the past 40 years (e.g. Piotrowski et al. (2011)). In parallel it was cast in the modern Fortran, following third party coding standards, (Ziemiański et al., 2021), or re-created from documentation using modern C++ (Jaruga et al., 2015).

to facilitate code portability across architectures. In the following sections we discuss two example applications of the eDSL concept to C/C++ and Fortran codes, using CUDA and Kokkos backends to interface with GPU accelerators.

2 Application agnostic eDSL for accelerators

Many processes in Earth System Science (ESS) are mathematically modelled with conservation and transport laws in the form of systems of partial differential equations. These systems are solved typically with a handful of discretisation approaches, e.g. finite differences, finite volumes or finite elements. This consistency gives rise to a core implementation and computational structure which is rather common across various ESS domains. Moreover, the computational challenges and bottlenecks are overlapping and analogous, which encourages formalization of the common aspects into an abstraction layer, leading to the concept of Domain Specific Languages (DSLs). Prominent examples for DSLs abstracting solutions for partial differential equations with various complexity are the Unified Form Language (UFL, Alnæs et al. (2014); Rathgeber et al. (2016)), the library framework OP2 for solving unstructured mesh-based applications (Mudalige et al., 2012; Balogh et al., 2018), and the ExaStencils code generation framework (Lengauer et al., 2014, 2020). Because of the similar numerical and computational structure, many geoscientific codes can benefit from a common DSL approach (Lawrence et al., 2018; Louboutin et al., 2019). Here, we argue that an embedded DSL (eDSL) is a suitable solution to ensure performance portability (and future-proofing) of geoscientific codes. We aim to demonstrate that a (rather simple) lightweight approach based on macro/preprocessor code creation for a given hardware configuration may already result in significant, worthwhile performance gains. From now on, for brevity we use the eDSL term in the context of such minimal and lightweight DSL approach, and some claims may specifically address this particular subset of a family of DSL approaches.

Computationally intensive kernels tend to repeat the same set of operations for large sets of cells, or they may usually be made such with the separation of a single dimension. In Earth System modelling, the latter typically happens where a sequential and horizontally independent physical process (e.g. sedimentation) occurs in an atmospheric column. Consequently, it is the kernel control execution structure (e.g. loops) that needs to be generalized to enable porting to a broad range of architectures and programming models. The eDSL approach allows a lightweight and minimally invasive intervention into the existing code structure. The key code modification is wrapping kernels with an abstraction layer in the form of preprocessing directives introducing sets of loops or delegating the execution via lambda functions. Here we argue that it helps to minimize the (apparent) cost function which relates to the compromise between productivity, portability and performance (Lawrence et al., 2018). Evidently, by enabling different carefully-crafted backends (e.g. CUDA, OpenMP, OpenACC, Fortran CUDA Loop Directives (CUF), HIP, Kokkos) performance-portability can be achieved, at least to a reasonable degree. Code readability is minimally affected, as only localized kernel (execution) definitions need to be implemented or modified, whereas the kernel body remains in place. Developer productivity is not impaired, since there is little new high-level (frontend) code to become familiar with. Conversely, readability and productivity may be potentially improved, because lower-level software engineering code is hidden and, optionally, abstracted loop names may provide extra annotation. Finally, the eDSL facilitates encapsulation and the separation-of-concerns (Bauer et al., 2021; Dauxois et al., 2021), allowing for easier maintenance and code sustainability. In other words, for domain scientists, the eDSL approach keeps the frontend code clean of hardware specific code and optimizations, and for software engineers, keeps backend code at a higher abstraction level.

The eDSL is better understood as a framework (or approach) rather than a library. This makes it flexible and agnostic of the specific application, in the sense that the abstraction itself can be implemented for any particular application. The prerequisite is to have the critical kernels and the structure of their control flow statements identified and parallelism is already highly and well exposed. For example, Eulerian models typically solve PDEs by involving gradient or flux computations across cell edges or stencils, cell updates, matrix assemblies and so on. Consequently, a few types of parallel loops can be identified, and encapsulated by the eDSL, gradually targeting the key kernels first. Obviously, despite conceptual and algorithmic similarities, the design of a particular application varies in the choice of programming languages used for implementation. Furthermore, it may have quite diverse data structures and workflows, for which the eDSL needs to be tailored for. In a nutshell, we presume that the eDSL concept can be straightforwardly implemented on different frontend scientific codes. However, it requires a *flavour* catering for the specific high-level language and the lower-level syntax and technicalities of the targeted backends.

Finally, the lightweight eDSL is self-contained within the high-level scientific application code, and not an external dependency. This minimizes the issues of building with yet-one-more dependency, while also allowing the flexibility of not relying on the evolution and sustainability of any given dependency. It also facilitates its use for small-scale codes which may not be typically deployed in HPC centres, but on small-scale clusters (and even single consumer-grade GPU devices) with a limited software stack management.

Herein, we present and discuss two independent applications of C and Fortran-based eDSLs for performance portability of well-established geoscientific codes that are ParFlow and EULAG, respectively. Both codes rely on MPI distributed memory parallelism, whereas the intra-device parallelism is potentially delegated to the DSL layer. The architecture-specific code that is dealing with memory management and compute kernels is embedded into the eDSL preprocessor macros. As a result, the macro definition looks different on the chosen architecture/programming model, however, the actual code duplication is minimal as compared to the multiple-codebase approach. In the following sections we demonstrate the aforementioned advantages of the approach.

3 ParFlow eDSL for accelerators

ParFlow is a widely adopted hydrologic model with the history of development spanning over the last three decades (Kuffour et al., 2020). Typical applications include modeling groundwater and overland flow using finite difference and finite volume schemes on a regular Cartesian grid with fully implicit time integration (Woodward, 1998; Kollet and Maxwell, 2006; Maxwell, 2013). The resulting non-linear coupled system of equations is solved using Newton-Krylov methods and multigrid preconditioning.

In order to separate the hardware architecture complexities from the scientific numerical model, ParFlow has adopted an eDSL based on C preprocessor macros (ParFlow eDSL). This domain-specific language provides an API for all necessary operations, such as memory management, message passing, and looping.

3.1 Kokkos implementation

The first backend option for GPU support in ParFlow was based on CUDA (Hokkanen et al., 2021), which shows excellent performance and scaling. Conveniently, adding backend support for libraries such as Alpaka (Zenker et al., 2016), Kokkos (Edwards et al., 2012), and RAJA (Beckingsale et al., 2019) to target more architectures (and, thus, performance portability) is straightforward and does not require any major changes. This is because, in ParFlow, the approach of passing loop contents form a lambda function to a general CUDA kernel can also be used with the aforementioned libraries as demonstrated for the ParFlow Kokkos backend in this study. Thanks to the macro-based abstraction layer, the Kokkos backend is not a compulsory dependency for ParFlow. This is important to hedge the risk of introducing third-party dependencies of unknown sustainability. Also, the Kokkos API is accessed through only a few ParFlow eDSL macros, such that replacing Kokkos with another similar backend is easy (essential in case Kokkos development stagnates).

The ParFlow GPU implementation heavily leverages Unified Memory, using the same pointers in both: host and device codes. As compared to the manual memory management, Unified Memory significantly decreases the development effort and results in clearer and simpler code. Furthermore, in the GPU implementation, a pool allocator for Unified Memory is used to improve the performance (Hokkanen et al., 2021).

In our experience, the first backend option (CUDA) required several months of development time from a single developer. Adding Kokkos as an alternative to CUDA required less than two weeks of additional full-time work with no prior knowledge of Kokkos. Compared to CUDA, Kokkos lacks a C interface, thus the Kokkos API calls must be placed into a separate C++ compilation unit that provides wrapper functions callable from Parflow C code.

In terms of memory management, Kokkos introduces Kokkos View, a multidimensional array template. Views intend to encapsulate low-level architecture-dependent choices on memory layouts, while enabling fine-grained control of the memory space (e.g., host, device or unified memory) where the data lives. Of course, the backend implementation determines how data Views are translated into hardware specific memory allocations (e.g., cudamalloc). Although Views are indeed a convenient data type, they are not used in the ParFlow Kokkos backend. This is because changing the variable types from raw pointers to Kokkos Views in a code that is expected to compile with a C compiler (if Kokkos is not used), would entail unwanted rewrites, wrappers, or syntactic changes widely throughout the codebase and cause undesirable increased complexity in the implementation. Fortunately, the use of Kokkos Views is optional, and memory can be allocated using the *Kokkos::kokkos_malloc* function. When called without template arguments, this functions allocates into the Kokkos default memory space, which depends on the Kokkos backend chosen during compilation time. An instructive example of allocating and deallocating a vector type with ParFlow eDSL is given in Listing 1. The macro definitions for the native host and Kokkos memory allocations and deallocations are given in Listings 2 and 3.

Listing 1. ParFlow eDSL dynamic memory allocation and deallocation.

```
/* Vector allocation */
vector = talloc(Vector, 1);

/* Vector deallocation */
tfree(vector);
```

Listing 2. ParFlow eDSL macro definitions for native host memory allocation and deallocation.

```
/* The macro definitions are placed in a header file (API) */
#define talloc(type, count) \
(type*)malloc(sizeof(type) * (unsigned int)(count))

#define tfree(ptr) free(ptr)
```

Listing 3. ParFlow eDSL macro definitions for Kokkos memory allocation and deallocation.

```
/* The macro definitions are placed in a header file (API) */
#define talloc(type, count) \
(type*)_kokkos_malloc(sizeof(type) * (unsigned int)(count))

#define tfree(ptr) _kokkos_free(ptr)

/* The Kokkos wrapper functions are placed in a cpp compilation unit */
#include <Kokkos_Core.hpp>
extern "C"{
  void* _kokkos_malloc(size_t size){
    return Kokkos::kokkos_malloc(size);
  }
  void _kokkos_free(void *ptr){
    Kokkos::kokkos_free(ptr);
  }
}
```

The implementation of compute kernels in ParFlow follows an approach similar to the memory management. However, there can be multiple compute kernels that often consists of different logic and varying number of variables required in the calculations. Thus, to address a wide range of scenarios, the loop execution is defined by preprocessor macros, which take the loop body as a macro argument. The loop body is typically provided as the last argument to the macro, e.g., in Listing 4 the loop body refers to the contents within the curly brackets. Providing the loop body as a macro argument is key in allowing to use the same macros for many compute kernels regardless of the loop logic and the number of variables involved in the calculations.

Similarly to the memory management, the loop macros take different form while aiming at the host or the device execution. For illustration, the loop macro invocation in Listing 4 is expanded for a sequential execution on the host and for a parallel execution using Kokkos in the form of Listings 5 and 6, respectively. For sequential execution, macro injects the loop body macro argument inside the innermost loop in the macro definition. Alternatively, in case of the Kokkos backend, the loop body forms a lambda function that is passed to the Kokkos kernel, where the lambda function simply captures all required variables by their value (i.e., with pointers the captured value is just the address the pointer is pointing to, and the data is accessed by denoting the offset with the conventional square bracket syntax).

Listing 4. BoxLoopI0: a simple loop over the discretized domain.

```
double *fp;
double *pp;
double value;
Subvector *f_sub;

/* some code missing here */

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
{
  int ip = SubvectorEltIndex(f_sub, i, j, k);
  fp[ip] = pp[ip] - value;
});
```

Listing 5. BoxLoopI0 macro definition for sequential execution on the host.

```
#define BoxLoopI0(i, j, k, ix, iy, iz,                              \
  nx, ny, nz, loop_body)                                            \
{                                                                   \
  for (k = iz; k < iz + nz; k++)                                    \
    for (j = iy; j < iy + ny; j++)                                  \
      for (i = ix; i < ix + nx; i++)                                \
      {                                                             \
        loop_body;                                                  \
      }                                                             \
}
```

Listing 6. BoxLoopI0 macro definition for parallel execution using Kokkos.

```
#define BoxLoopI0(i, j, k, ix, iy, iz,                            \
  nx, ny, nz, loop_body)                                          \
{                                                                 \
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)           \
  {                                                               \
    i += ix; j += iy; k += iz;                                    \
    loop_body;                                                    \
  }                                                               \
                                                                  \
  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});        \
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);                 \
}
```

3.2 ParFlow performance results

The performance of the Kokkos implementation (using CUDA through Kokkos) was evaluated on the Booster module of the JUWELS supercomputer (Jülich Supercomputing Centre, 2019) where each node has dual AMD EPYC Rome 7402 processors (2×24 cores @ 2.8 GHz) and 4 NVIDIA A100 40 GB GPUs. The nodes are connected through 4 HDR200-InfiniBand devices. The reference results were obtained by performing the simulation without accelerator devices on all available CPU cores. The GPU runs only need a single process per GPU, therefore 4 CPU cores per node were used (with 4 available GPUs).

The benchmark consists of a variably saturated infiltration problem into a homogeneous soil with a fixed water table at a depth of 6 m, and a constant infiltration rate of 8×10^{-4} m / hour. The vertical and lateral spatial discretization was 0.025 and 1 m, respectively. The time step size was 1 h. The profile was initialized with a hydrostatic profile based on a matric potential of -9 m at the top. This results in a considerable initial hydrodynamic disequilibrium with respect to the water table at the bottom boundary. The number of grid cells in the lateral directions was varied to change the total number of degrees of freedom in performance testing (weak scaling).

Figure 1 shows the performance gain from GPUs for a single node. The horizontal and left vertical axes represent the problem size and the *performance*, respectively, where the latter describes the number of cells per second. The performance with GPUs is plotted using CUDA directly with pool allocation (no Kokkos), and using CUDA through Kokkos with and without pool allocation. For details on the Unified Memory pool allocation the interested reader is referred to (Hokkanen et al., 2021). Note, no pool allocation was used on the CPU.

Using Kokkos without CUDA-specific code results in a 20% performance reduction for the largest ParFlow problem size that fits on a single GPU, when compared with the CUDA implementation. This is mainly caused by parallel reductions, array initializations, and usage of pinned host/device memory for MPI staging buffers to enable GPU-direct P2P communication with Remote Direct Memory Access (RDMA) when using Kokkos. The latter two overheads can be easily resolved

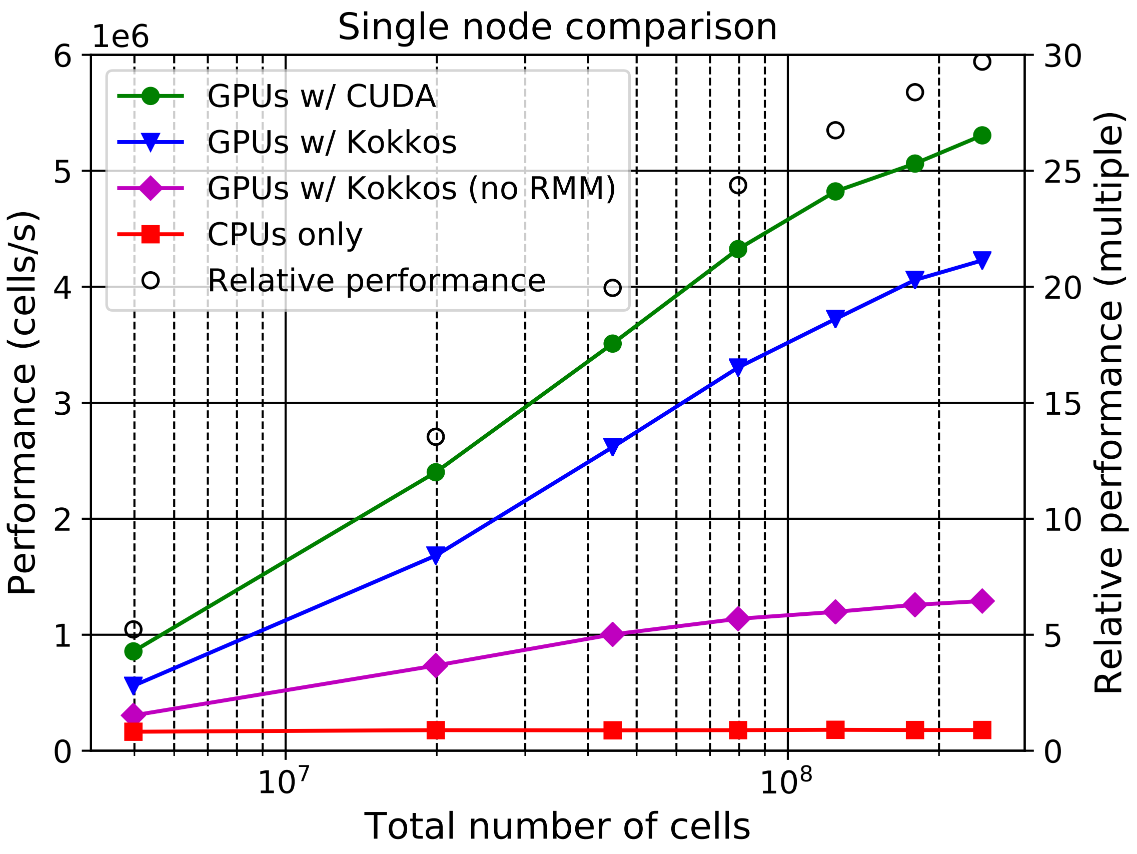

Figure 1. Parflow single node performance comparison. Relative performance refers to the performance ratio between CUDA and CPU implementations of the code. RMM denotes Rapid Memory Manager.

by using CUDA-specific function calls or template arguments with the Kokkos library, which, however, leads to an undesired non-architecture-agnostic implementation. Furthermore, the Unified Memory pool allocation introduces further architecture-specificity, as the chosen memory manager (library) only supports CUDA. Nonetheless, viable alternatives exist. An example is Umpire (Beckingsale et al., 2020), which supports heterogeneous architectures, as well as multiple backends (e.g., CUDA and HIP). This is very relevant, since the impact of the pool allocator is significant and more than triples the performance for the biggest problem sizes. The performance boost of the pool allocator is similar to the one resulting from using directly the CUDA backend (Hokkanen et al., 2021). This is explained by the recurring Unified Memory allocations and deallocations during the simulation in ParFlow.

Relative performance, which is the ratio between the accelerated and non-accelerated simulation when using CUDA directly, is given as circles in Figure 1. The relative performance increases from ∼5 to ∼30 with increasing problem size. The performance on CPU may be improved via pool allocation in future, which may reduce the performance increase.

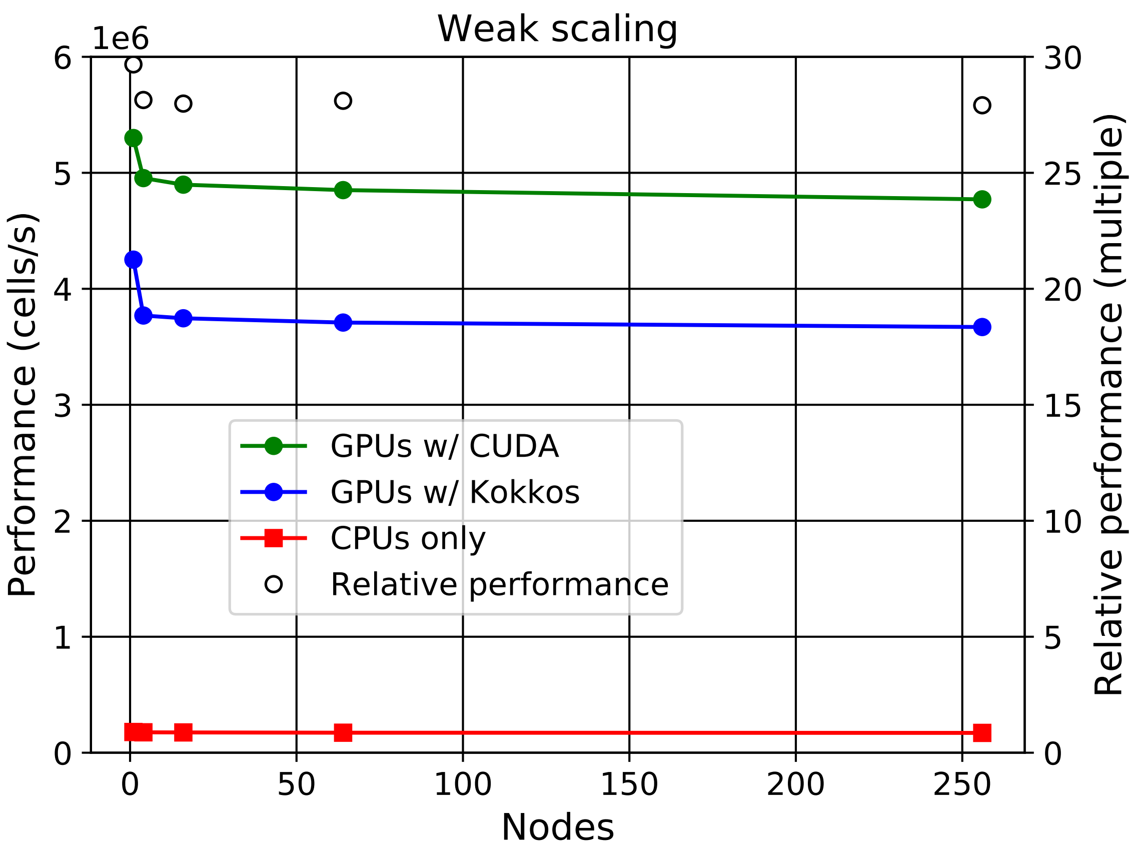

Figure 2. Weak scaling comparison.

Figure 2 represents weak scaling for 1, 4, 16, 64, and 256 nodes using the largest problem size from Figure 1. The relative performance when directly using CUDA saturates at ∼28 when increasing the number of nodes which suggests good weak scaling behavior and performance. In comparison, the performance achieved with the generic Kokkos backend is about 20-25% worse.

4 EULAG/MPDATA Fortran eDSL

To verify if the C/C++ eDSL concepts are applicable to Fortran, a similar eDSL was developed for the MPDATA advection (Smolarkiewicz (2006)) dwarf, cf. (Müller et al. (2019), Rojek et al. (2017) and references therein). Belonging to the class of iterated non-linear upwind schemes, this benchmark well balances the compactness and complexity. MPDATA constitutes a conservation-law form of the generalized transport equation for a specific variable ψ (e.g. velocity component, potential temperature, Exner pressure, moist species, passive tracers, etc.):

$$\frac{\partial \mathcal{G}\varrho\psi}{\partial \bar{t}} + \overline{\nabla}\cdot(\mathcal{G}\varrho\mathbf{v}\psi) = \mathcal{G}\varrho R\ . \tag{1}$$

where $\mathcal{G}$ is the flow Jacobian, $\overline{\nabla}$ denotes vector of partial derivatives over the system of arbitrary curvilinear coordiantes, ϱ denotes density, $\mathbf{v}$ is the transporting velocity and R is the source/sink term. Notably, MPDATA is an integral part of the EULAG (Eulerian/semi-Lagrangian fluid solver) dynamical core, and is employed by a number of scientific (Prusa et al. (2008); Jaruga et al. (2015)) and weather codes (Kühnlein et al. (2019); Ziemiański et al. (2021)). As part of the EULAG dynamical core, MPDATA advection is orchestrated with the preconditioned Krylov solver for pressure and explicit diffusion operator. Extension of the proposed concept to a full Navier-Stokes solver is straightforward, however, out of the scope of this study.

4.1 CUDA Fortran backend implementation

Herein, this work we focus on a subset of possible hardware backends, targeting the large and (relatively) low-hanging performance gain of code portability to both: CPU and NVIDIA GPU using CUDA Fortran (CUF) directives. In the simplest case, e.g. within the pure-MPI code it may appear superfluous to abstract loops rather than insert CUF directives directly. However, the abstraction allows to efficiently hide the code optimization that otherwise would make the code unreadable and unmaintainable. Furthermore, for the direct porting of code to different CPU compilers, some preprocessing is unavoidable. For example, the MANAGED or DEVICE attributes that are needed to exploit the CUDA Unified memory interface are currently not recognized by other than PGI/NVIDIA Fortran compilers. Last but not least, mixing several sets of precompiler directives heavily impairs the code readability.

The memory allocation for the CUF backend, similar to ParFlow, leverages Unified Memory. The necessary attribute is introduced at variable declaration stage with the preprocessor definitions given in Listing (7). Note that to assure for minimization of the host-device memory transfers, it is beneficial to specify auxiliary (temporary) variables explicitly as DEVICE variables. Spurious memory page faults, in theory, can be prevented by using CUDA memory hints (or prefetching, if applicable), but we were unable to avoid the unnecessary transfers when declaring temporary variables as MANAGED. Re-definition of the variable declaration needs to be performed consistently throughout the code, which is probably the most time-consuming effort in the implementation of the eDSL.

Listing 7. Fortran eDSL memory allocation abstraction

```
#ifdef CUDACODE
#define    REAL_sp        REAL(KIND=sp), MANAGED
#define DEVREAL_sp     DEVREAL(KIND=sp), DEVICE
#else
#define    REAL_sp        REAL(KIND=sp)
#define DEVREAL_sp        REAL(KIND=sp)
#endif
```

The key practical prerequisite for the readable abstraction of loops is to identify the number and type of possible loop configurations. This is of particular importance for the finite-difference/finite-volume fluid solvers, where operations can be carried out on several (e.g., so called staggered "C"/unstaggered "A") grids. Moreover, the solvers often employ specialized stencils along the domain's boundaries. As opposed to the ParFlow eDSL, we choose not to pass the loop extents as a macro parameter. Conversely, aiming at the meaningful code annotation, we let the abstract loop names correspond to the geometric/topological type of operation performed, e.g. "XYZFullDomainAgrid". In turn, the definition of a particular GPU kernel for each particular kind of loop is provided by the macro. The loop definition is injected into the code, along with the appropriate CUDA Fortran directive, which is translated by the NVIDIA compiler to an actual accelerator kernel. The simplest three-dimensional example of such macro providing iteration on the full computational domain is presented in Listing (8).

Listing 8. Fortran eDSL loop abstraction

```
#define FullXYZDomainLoopDC(loop_body)\
!$cuf kernel do(3) <<< (*,*), (32,4) >>>\
    do k=1,lp;\
      do j=1,mp;\
        do i=1,np;\
          loop_body\
        end do;\
      end do;\
    end do;
```

If more general macro definitions are needed, or a precise macro name reflecting the particular operation scope cannot be found, the formulation of Listing (8) to match Listing (4) can be extended in a straightforward fashion. Furthermore, this approach may be easily extended, e.g. to support overlapping computations and MPI communication or using multiple streams.

In a previous, currently unpublished development effort, the MPDATA dwarf was implemented with the GridTools DSL (Afanasyev et al. (2021)). As compared to GridTools, the first major advantage of a lightweight eDSL is that a rewrite from Fortran to C++ is not required. Moreover, the stencil definition is retained (in place) instead of being delegated to a separate structure (cf. Fig. 2 in Afanasyev et al. (2021)), which preserves excellent code readability and unobstructed debugging. A downside of the eDSL is that automated composing of stencil operations into multistages with corresponding implicit optimizations (cf. Fig. 3 and discussion in Afanasyev et al. (2021)) is not available and is delegated to the developer. However, in our experience the major prerequisite (to any eDSL approach) is judicious coding of special stencils realizing boundary conditions, which otherwise limit the (often essential for performance) composition of subsequent stencils into a single compute-intensive kernel. Thus, automated composition of stencils is often dictated by the numerics, regardless of the particular DSL approach. Finally, compiler overhead for the eDSL is negligible, whereas the compilation time of the full C++ DSL, depending on the language standard, may easily be a hundred times longer due to metaprogramming expenses. This is especially pronounced when implementing advection of multiple variables, e.g. realizing the timestep of the weather solver.

4.2 Fortran MPDATA performance results

In the performance analyses, the proposed test setup follows (Jaruga et al. (2015)), where the accuracy of the 3D MPDATA scheme is benchmarked in the rotating spherical passive tracer scenario. The dimensionless domain size $dx00$ is prescribed as $dx_{00} = 100^3$ (uniform in all three directions) and consists of an arbitrary $n \times m \times l$ spatial gridpoint distribution. The spherical tracer of initial radius $r = 0.15dx_{00}$ and constant magnitude $h = 4$ revolves around the point

$$(x_c, y_c, z_c) = \left(\left(0.5 - 0.25/\sqrt{3}\right)dx_{00}, \left(0.5 + 0.25/\sqrt{3}\right)dx_{00}, \left(0.5 + 0.25/\sqrt{3}\right)dx_{00}\right) \tag{2}$$

transported with constant angular velocity $\Omega = (0.1, 0.1, 0.1)/\sqrt{3}$. The advection velocity is defined on the C-grid in the form of non-dimensional Courant numbers, numerically averaged from the analytical definition on the A-grid defined in the point (x,y,z) as

$$(u, v, w) = 0.1/\sqrt{3}(-(y - y_c) + (z - z_c), (x - x_c) - (z - z_c), -(x - x_c) + (y - y_c)) \tag{3}$$

To verify the implementation accuracy, the uniform grid spacing $n = m = l = 59$ and timestep $dt = 0.036$ were tested in double precision for $nt = 556$ time steps needed to complete one revolution on the circular trajectory. The L_2 norm r of the tracer solution ψ is defined as

$$r = \frac{1}{nt \cdot dt}\sqrt{\frac{\sum_{i=1,nml}\left(\psi_{exact} - \psi\right)^2)}{nml}} \tag{4}$$

where $nml = 59^3$ and ψ_{exact} represents analytical solution. For both: legacy and optimized codes and single and double precision, $r_{nt=556} = 0.028$, which matches the result of Jaruga et al. (2015).

For the purpose of scalability evaluation, the advection procedure is invoked in a loop over 500 timesteps. The GPU implementation is executed on the 2x2x1 MPI grid, whereas the CPU implementation is decomposed into the 3x4x4 MPI grid to minimize the total transmitted amount of halo data (Piotrowski et al. (2011)). All tests discussed rely on non-blocking CUDA-aware MPI implementation. Similar to Figure 1, the performance gain for 4 x GPU versus 2 x CPU on a single node on JUWELS Booster as a function of problem size is shown in Figure 3 and demonstrates close to $\approx$ 18-fold speedup for the largest problem sizes. Performance of the GPU implementation improves significantly with the number of cells until a total of $\approx 8 \cdot 10^7$ cells are reached. The CPU performance diminishes at the largest problem sizes (probably due to the worsening cache utilization), whereas the GPU computing capability seems to gradually saturate. Note that in this comparison both: CPU and GPU computational formulation deserve extra optimization effort, that is beyond the scope of this study. For example, CPU formulation would normally benefit from the hybrid MPI-shared parallelization and from loop-tiling optimizations. Furthermore, the GPU performance would be typically limited by total memory available on the node-set of GPUS. Last but not the least, ratio between number of GPUs to number of CPUs varies between supercomputing architectures.

For most atmospheric applications considered by the EULAG community, where the temporal integration lasts thousands to millions of timesteps, the key metric is time-to-solution, thus, scalability limits at very large domain sizes are of less importance. For this reason, strong scaling is usually of more interest metric than a weak scaling. To investigate strong scalability limits,

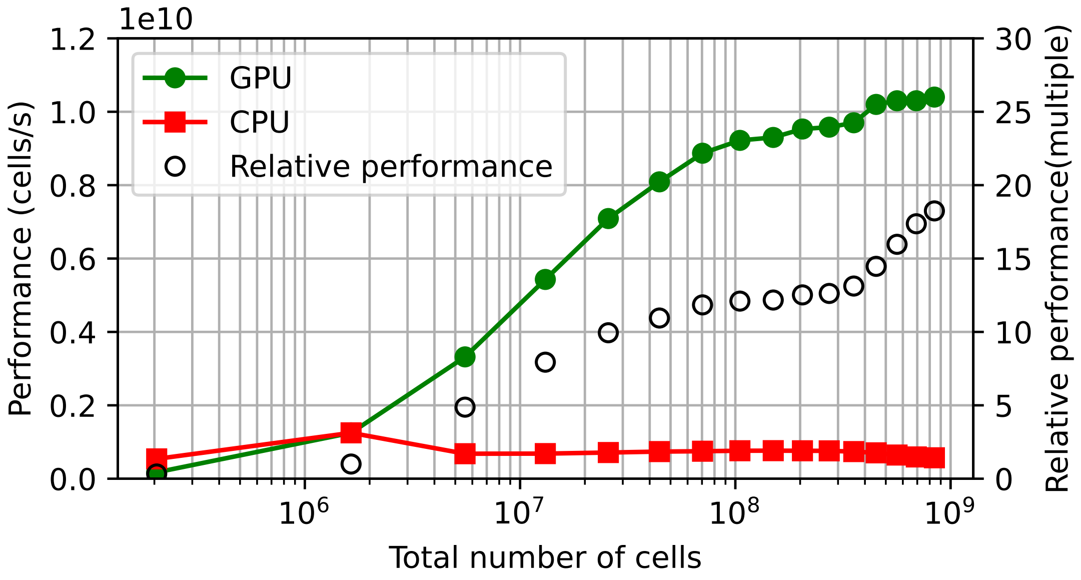

Figure 3. MPDATA single node performance comparison.

timings of integrating MPDATA advection for one full revolution of the sphere with $7-, 14-and\ 21-$ times refined resolution are presented in Fig. 4. Guided by the inspection of Figure 3 we expect that already at the smallest problem size of 413^3 gridpoints the GPU performance is close to saturation. Therefore, we can't expect perfect scalability when under-loading the GPU device. Indeed, the 413^3 does not scale well beyond the single node, whereas for the largest problem size of 1239^3 the strong scalability weakens at the 16 GPU nodes. CPU scalability is better at the higher node counts, however, consistently delivers longer time-to-solution than a GPU-based integration.

Notwithstanding the operational requirements to limit the time-to-solution, weak scalability study reveals a good parallel capabilities up to 12000 CPU cores/1000 GPUs. Apart from the small node counts, the speed-up of GPU over CPU computation is consistently close to 11.

5 Prerequisites, principles and restrictions of a lightweight eDSL application

It is difficult to define the universal but detailed recipes for minimally invasive eDSL applications across the variety of computational fluid dynamics and geoscientific codes. However, it is certainly possible to name general rules and concepts for the development of such eDSLs, which are enumerated and briefly discussed below.

1. Major code sections should be reasonably separated to aid gradual eDSL application.

   The most consuming (implementation-wise) part of the eDSL, at least from the Fortran perspective, is the introduction of the abstracted variable declarations throughout the code. It is therefore beneficial to conceptually separate CPU-specific compilation units (e.g. initialization, I/O and finalization) from the actual computations on the accelerator. Code

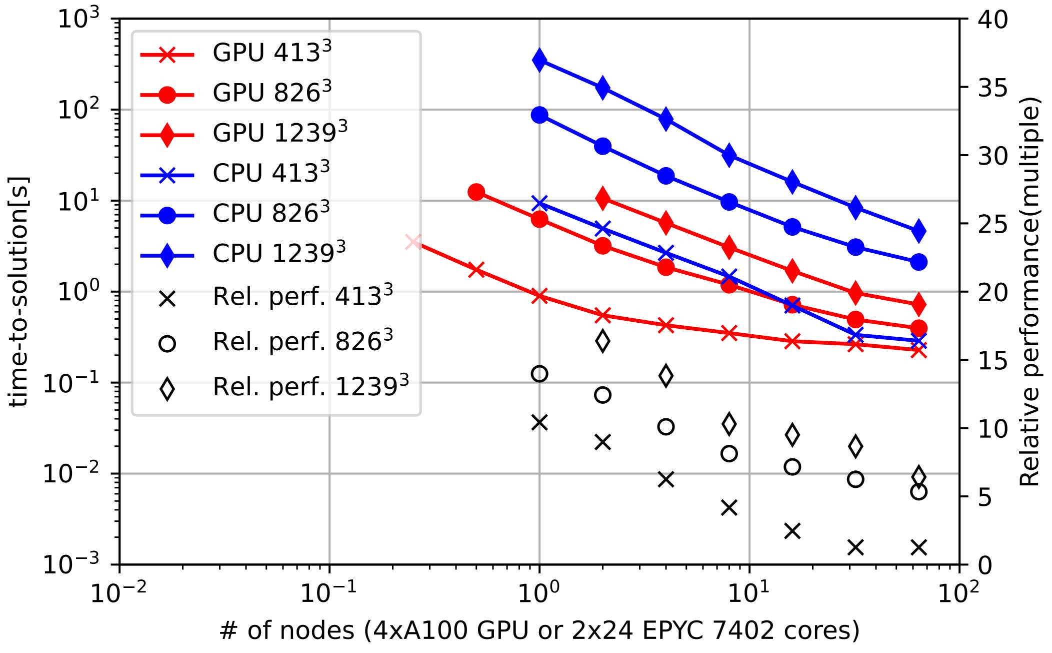

Figure 4. Strong scalability of MPDATA gauge implementation for different problem sizes. For time-to-solution (colored lines) the leftmost datapoints correspond to 1/2 and 1/4 of a 4 GPU node, i.e. 2 or 1 NVIDIA A100 GPU, respectively. Relative performance gain (point symbols) is shown for full node GPU vs. CPU simulations. Numbers in the legend denote total cell counts for the refined reference domain: $(7*59)^3$, $(14*59)^3$ and $(21*59)^3$.

modularization, as applied extensively in modern geoscientific modeling frameworks (Shrestha et al. (2014); Gasper et al. (2014); Pham et al. (2021)), is a prerequisite to achieve performance portability for large code bases.

2. Exposition of parallelism and refactoring towards increased computational intensity should be performed before the eDSL application. For stencil algorithms on legacy architectures, typically memory(cache) bandwidth-bound, significant improvement in the computational performance may be achieved already by code restructuring. For example, the optimized (CPU) MPDATA dwarf version evinces typically about two times better performance as compared to the legacy formulation. The key optimization here was loop fusion enabled by the removal of several intermediate memory stores to auxiliary variables, thus increasing arithmetic intensity. One could argue that such operations should rather be performed by the DSL itself. However, in presence of elaborated boundary conditions allowing for multiple scenarios, this is often not easy, as computation of boundary stencil (e.g. zero flux, zero divergence, etc.) prevents naive kernel merging. Treatment of the boundary conditions needs thus be judiciously planned before any eDSL (or DSL) approach is applied. On the other hand, highly-optimized single-application codes may benefit from the custom-tailored DSL that handles effectively only a subset of possible b.c. realizations.

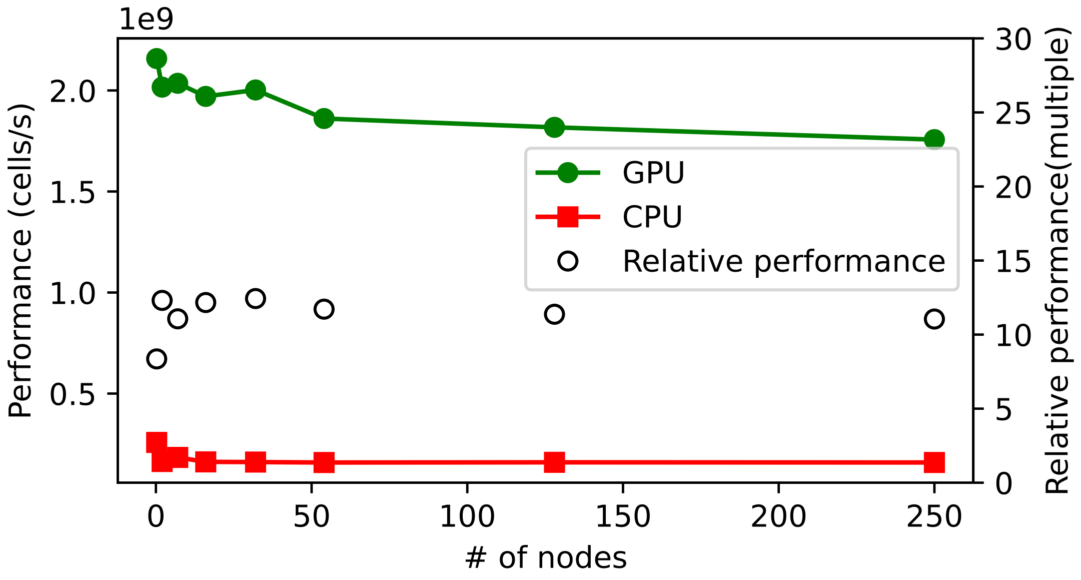

Figure 5. Weak scalability of MPDATA gauge implementation for the 492^3-gridpoints-per-node problem size.

3. Critical sections requiring code specialization should be reorganized to minimize code duplication.

   It is not possible to realize optimal computational formulations for all architectures within a single implementation. For example, some optimizations, especially in physical parameterizations, prove effective for GPUs, but have adverse impact on CPU performance. This aspect may be to some degree circumvented by specialization depending on the architecture, but ultimately feasibility of such approach depends on the particular application. However, carefully designed loop abstraction in the eDSL layer should at least be able e.g. to realize code specialization involving loop reordering, merging and tiling without code duplication.

4. Non-standard order of stencil execution in the domain may be considered to improve parallel performance.

   Abstraction of loops in a eDSL, next to tiling or reordering, enables silent generation and prioritization of loops realizing inner domain or (subdomain) boundary computations. The computation of stencils at the subdomain boundaries can be thus prioritized to realize non-blocking communication, overlapped with computation inside of a subdomain, without the loss of readability.

5. For sufficiently large computational grids, a memory reuse strategy is needed to fit inside the accelerator device.

   Given the enormous computing power of accelerators, it is necessary to feed them with large enough computational tasks, which is usually linked to large memory demands. For example, extending the MPDATA eDSL to the full EULAG fluid solver may lead to GPU memory limitations requiring the implementation of a memory pool for auxiliary variables, which has been discussed in section 3.

6 Conclusions

In recent decades, the scientific community faced the end of the Dennard scaling and the onset of the massively parallel computing era with growing heterogeneity of computing paradigms and architectures. The advent of the widely available accelerated supercomputing architectures modified the landscape of computational geophysics even further. The pioneers of computing on GPU and x86 manycore processors paved new routes towards fast and energy efficient calculations. In this work, we propose a lightweight embedded Domain Specific Language (eDSL) approach for performance portability. We report our experience to the still largely unclaimed land of performant execution of geoscientific applications on accelerators. We argue that thanks to similarities across the Earth System Science codes and their implementations it is possible to define general concepts of the eDSL approach, greatly facilitating performance portability. Furthermore, the approach seems to be easily extendable to the general field of Computational Fluid Dynamics and beyond.

Based on the proposed lightweight eDSL concept, we presented results of accelerator ports of the two established geophysical research codes ParFlow and MPDATA. These codes span exemplarily a range of applications covering transport processes in the Earth System from the cloud microscale, through variably saturated ground water and surface water flows, to global atmospheric circulation. The study demonstrates that many geophysical research applications, which for various reasons can not rely on a large team of software engineers, nevertheless may gain access to rapid computational benefits when applying the proposed eDSL concept. At the same time, code readability is maintained and established coding paradigms remain accessible to the domain scientists.

From our experience, developers of geophysical research codes need to balance between the needs of the domain scientists and an appropriate use of (finite) supercomputing resources under the common conditions of tight research time frames. We propose to consider the presented eDSL concept as a lightweight approach towards legacy code performance portability on diverse supercomputing architectures, avoiding abrupt transition to new coding paradigms and large human engineering effort, preserving flexibility and remaining hardware and vendor agnostic. The approach would rarely offer ways to approach the hardware performance limits, such as FLOP-based peak performance or saturated memory bandwidth. Nonetheless, it does offer potentially large speed-ups. These may be close to the memory-bandwidth ratio of the full-node combined GPU to CPU transfer capabilities, or even more in case of compute-bound code. Moreover, the eDSL enables the possibility to harness efficiently upcoming hardware on supercomputing systems in the exascale range. In our opinion the eDSL concept will be applicable to the majority of non-time critical applications that constitute the foundation of modern computational geosciences.

Code availability. The current version 3.12 of Parflow is available from the project website: https://parflow.org under the licence GNU LGPL. The exact version of the model used to produce the results used in this paper is archived on Zenodo (https://zenodo.org/record/4957977). MPDATA eDSL 1.0 is available from https://github.com/piotrows/advectionHPCtester. The exact version of the model used to produce the results used in this paper is archived on Zenodo (https://zenodo.org/record/8178549).

Author contributions. SK is responsible for conceptualization and supervision. SK, OS and DCV contributed to the funding acquisition and project administration. JH and ZP designed methodology, software design and conducted investigation process along with validation and visualization. ZP and JH prepared the original draft. ZP, SK, OS and DCV contributed to preparation of final manuscript.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. The authors gratefully acknowledge the Earth System Modelling Project (ESM) for funding this work by providing computing time on the ESM partition of the supercomputer JUWELS at the Jülich Supercomputing Centre (JSC). This work received funding from the Initiative and Networking Fund of the Helmholtz Association through the projects ‘Advanced Earth System Modelling Capacity (ESM)’ and ‘Pilot Lab Exascale Earth System Modelling (PL-ExaESM)’. This work has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 824158 (EoCoE-2). Z.P. Piotrowski acknowledges support from the “Numerical weather prediction for sustainable Europe” project, carried out within the FIRST TEAM program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund. Commments from Piotr Smolarkiewicz and Michael Lange on earlier versions of this manuscript are gratefully acknowledged.

References

Afanasyev, A., Bianco, M., Mosimann, L., Osuna, C., Thaler, F., Vogt, H., Fuhrer, O., VandeVondele, J., and Schulthess, T. C.: GridTools: A framework for portable weather and climate applications, SoftwareX, 15, 100 707, https://doi.org/https://doi.org/10.1016/j.softx.2021.100707, 2021.

Alnæs, M. S., Logg, A., Ølgaard, K. B., Rognes, M. E., and Wells, G. N.: Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations, ACM Trans. Math. Softw., 40, https://doi.org/10.1145/2566630, 2014.

Balogh, G., Mudalige, G., Reguly, I., Antao, S., and Bertolli, C.: OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling, in: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 59–70, https://doi.org/10.1109/LLVM-HPC.2018.8639205, 2018.

Bauer, P., Dueben, P. D., Hoefler, T., Quintino, T., Schulthess, T. C., and Wedi, N. P.: The digital revolution of Earth-system science, Nature Computational Science, 1, 104–113, https://doi.org/10.1038/s43588-021-00023-0, 2021.

Beckingsale, D. A., Scogland, T. R., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., and Ryujin, B. S.: RAJA: Portable Performance for Large-Scale Scientific Applications, in: Proceedings of P3HPC 2019: International Workshop on Performance, Portability and Productivity in HPC - Held in conjunction with SC 2019: The International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 71–81, Institute of Electrical and Electronics Engineers Inc., https://doi.org/10.1109/P3HPC49587.2019.00012, 2019.

Beckingsale, D. A., McFadden, M. J., Dahm, J. P. S., Pankajakshan, R., and Hornung, R. D.: Umpire: Application-focused management and coordination of complex hierarchical memory, IBM Journal of Research and Development, 64, 00:1–00:10, https://doi.org/10.1147/jrd.2019.2954403, 2020.

Dauxois, T., Peacock, T., Bauer, P., Caulfield, C. P., Cenedese, C., Gorlé, C., Haller, G., Ivey, G. N., Linden, P. F., Meiburg, E., Pinardi, N., Vriend, N. M., and Woods, A. W.: Confronting Grand Challenges in environmental fluid mechanics, Physical Review Fluids, 6, https://doi.org/10.1103/physrevfluids.6.020501, 2021.

Edwards, H. C., Sunderland, D., Porter, V., Amsler, C., and Mish, S.: Manycore performance-portability: Kokkos multidimensional array library, Scientific Programming, 20, 89–114, https://doi.org/10.3233/SPR-2012-0343, 2012.

Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., Lapillonne, X., Leutwyler, D., Lüthi, D., Osuna, C., Schär, C., Schulthess, T. C., and Vogt, H.: Near-global climate simulation at 1km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0, Geoscientific Model Development, 11, 1665–1681, https://doi.org/10.5194/gmd-11-1665-2018, 2018.

Gasper, F., Goergen, K., Shrestha, P., Sulis, M., Rihani, J., Geimer, M., and Kollet, S.: Implementation and scaling of the fully coupled Terrestrial Systems Modeling Platform (TerrSysMP v1.0) in a massively parallel supercomputing environment – a case study on JUQUEEN (IBM Blue Gene/Q), Geoscientific Model Development, 7, 2531–2543, https://doi.org/10.5194/gmd-7-2531-2014, 2014.

Hokkanen, J., Kollet, S., Kraus, J., Herten, A., Hrywniak, M., and Pleiter, D.: Leveraging HPC accelerator architectures with modern techniques — hydrologic modeling on GPUs with ParFlow, Computational Geosciences, pp. 1–13, https://doi.org/10.1007/s10596-021-10051-4, 2021.

Jaruga, A., Arabas, S., Jarecka, D., Pawlowska, H., Smolarkiewicz, P., and Waruszewski, M.: libmpdata++ 1.0: a library of parallel MPDATA solvers for systems of generalised transport equations, Geoscientific Model Development, 8, 1005–1032, 2015.

Jülich Supercomputing Centre: JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre, Journal of large-scale research facilities, 5, https://doi.org/10.17815/jlsrf-5-171, 2019.

Kollet, S. J. and Maxwell, R. M.: Integrated surface-groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model, Advances in Water Resources, 29, 945–958, https://doi.org/10.1016/j.advwatres.2005.08.006, 2006.

Kuffour, B. N. O., Engdahl, N. B., Woodward, C. S., Condon, L. E., Kollet, S., and Maxwell, R. M.: Simulating coupled surface-subsurface flows with ParFlow v3.5.0: capabilities, applications, and ongoing development of an open-source, massively parallel, integrated hydrologic model, Geoscientific Model Development, 13, 1373–1397, https://doi.org/10.5194/gmd-13-1373-2020, 2020.

Kühnlein, C., Deconinck, W., Klein, R., Malardel, S., Piotrowski, Z. P., Smolarkiewicz, P. K., Szmelter, J., and Wedi, N. P.: FVM 1.0: a nonhydrostatic finite-volume dynamical core for the IFS, Geoscientific Model Development, 12, 651–676, https://doi.org/10.5194/gmd-12-651-2019, 2019.

Lawrence, B. N., Rezny, M., Budich, R., Bauer, P., Behrens, J., Carter, M., Deconinck, W., Ford, R., Maynard, C., Mullerworth, S., Osuna, C., Porter, A., Serradell, K., Valcke, S., Wedi, N., and Wilson, S.: Crossing the chasm: how to develop weather and climate models for next generation computers?, Geoscientific Model Development, 11, 1799–1821, https://doi.org/10.5194/gmd-11-1799-2018, 2018.

Lengauer, C., Apel, S., Bolten, M., Größlinger, A., Hannig, F., Köstler, H., Rüde, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., and Schmitt, C.: ExaStencils: Advanced Stencil-Code Engineering, in: Euro-Par 2014: Parallel Processing Workshops, edited by Lopes, L., Žilinskas, J., Costan, A., Cascella, R. G., Kecskemeti, G., Jeannot, E., Cannataro, M., Ricci, L., Benkner, S., Petit, S., Scarano, V., Gracia, J., Hunold, S., Scott, S. L., Lankes, S., Lengauer, C., Carretero, J., Breitbart, J., and Alexander, M., pp. 553–564, Springer International Publishing, Cham, 2014.

Lengauer, C., Apel, S., Bolten, M., Chiba, S., Rüde, U., Teich, J., Größlinger, A., Hannig, F., Köstler, H., Claus, L., Grebhahn, A., Groth, S., Kronawitter, S., Kuckuk, S., Rittich, H., Schmitt, C., and Schmitt, J.: ExaStencils: Advanced Multigrid Solver Generation, in: Software for Exascale Computing - SPPEXA 2016-2019, edited by Bungartz, H.-J., Reiz, S., Uekermann, B., Neumann, P., and Nagel, W. E., pp. 405–452, Springer International Publishing, Cham, 2020.

Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P. A., Herrmann, F. J., Velesko, P., and Gorman, G. J.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration, Geoscientific Model Development, 12, 1165–1187, https://doi.org/10.5194/gmd-12-1165-2019, 2019.

Maxwell, R. M.: A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling, Advances in Water Resources, 53, 109–117, https://doi.org/10.1016/j.advwatres.2012.10.001, 2013.

Mudalige, G., Giles, M., Reguly, I., Bertolli, C., and Kelly, P.: OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures, in: 2012 Innovative Parallel Computing (InPar), pp. 1–12, https://doi.org/10.1109/InPar.2012.6339594, 2012.

Müller, A., Deconinck, W., Kühnlein, C., Mengaldo, G., Lange, M., Wedi, N., Bauer, P., Smolarkiewicz, P. K., Diamantakis, M., Lock, S.-J., et al.: The ESCAPE project: energy-efficient scalable algorithms for weather prediction at exascale, Geoscientific Model Development, 12, 4425–4441, 2019.

Pham, T. V., Steger, C., Rockel, B., Keuler, K., Kirchner, I., Mertens, M., Rieger, D., Zängl, G., and Früh, B.: ICON in Climate Limited-area Mode (ICON release version 2.6.1): a new regional climate model, Geoscientific Model Development, 14, 985–1005, https://doi.org/10.5194/gmd-14-985-2021, 2021.

Piotrowski, Z. P., Wyszogrodzki, A. A., and Smolarkiewicz, P. K.: Towards petascale simulation of atmospheric circulations with soundproof equations, Acta Geophysica, 59, 1294, https://doi.org/10.2478/s11600-011-0049-6, 2011.

Prusa, J. M., Smolarkiewicz, P. K., and Wyszogrodzki, A. A.: EULAG, a computational model for multiscale flows, Computers & Fluids, 37, 1193–1207, https://doi.org/https://doi.org/10.1016/j.compfluid.2007.12.001, 2008.

Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., Mcrae, A. T. T., Bercea, G.-T., Markall, G. R., and Kelly, P. H. J.: Firedrake: Automating the Finite Element Method by Composing Abstractions, ACM Trans. Math. Softw., 43, https://doi.org/10.1145/2998441, 2016.

Rojek, K., Wyrzykowski, R., and Kuczynski, L.: Systematic adaptation of stencil-based 3D MPDATA to GPU architectures, Concurrency and Computation: Practice and Experience, 29, e3970, https://doi.org/https://doi.org/10.1002/cpe.3970, e3970 cpe.3970, 2017.

Shrestha, P., Sulis, M., Masbou, M., Kollet, S., and Simmer, C.: A Scale-Consistent Terrestrial Systems Modeling Platform Based on COSMO, CLM, and ParFlow, Monthly Weather Review, 142, 3466 – 3483, https://doi.org/10.1175/MWR-D-14-00029.1, 2014.

Smolarkiewicz, P. K.: Multidimensional positive definite advection transport algorithm: an overview, International Journal for Numerical Methods in Fluids, 50, 1123–1144, https://doi.org/https://doi.org/10.1002/fld.1071, 2006.

Sridhar, A., Tissaoui, Y., Marras, S., Shen, Z., Kawczynski, C., Byrne, S., Pamnany, K., Waruszewski, M., Gibson, T. H., Kozdon, J. E., Churavy, V., Wilcox, L. C., Giraldo, F. X., and Schneider, T.: Large-eddy simulations with ClimateMachine v0.2.0: a new open-source code for atmospheric simulations on GPUs and CPUs, Geoscientific Model Development Discussions, 2021, 1–41, https://doi.org/10.5194/gmd-2021-335, 2021.

Woodward, C. S.: A Newton-Krylov-muItigrid solver for variably saturated flow problems, Transactions on Ecology and the Environment, 17, 1998.

Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knupfer, A., Nagel, W. E., and Bussmann, M.: Alpaka - An abstraction library for parallel kernel acceleration, in: Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016, pp. 631–640, Institute of Electrical and Electronics Engineers Inc., https://doi.org/10.1109/IPDPSW.2016.50, 2016.

Ziemiański, M. Z., Wójcik, D. K., Rosa, B., and Piotrowski, Z. P.: Compressible EULAG Dynamical Core in COSMO: Convective-Scale Alpine Weather Forecasts, Monthly Weather Review, 149, 3563 – 3583, https://doi.org/10.1175/MWR-D-20-0317.1, 2021.