Geosci. Model Dev., 16, 2719–2736, 2023
https://doi.org/10.5194/gmd-16-2719-2023
© Author(s) 2023. This work is distributed under
the Creative Commons Attribution 4.0 License.

Geoscientific Model Development
Open Access
EGU

Development and technical paper

Pace v0.2: a Python-based performance-portable atmospheric model

Johann Dahm[1,★], Eddie Davis[1,★], Florian Deconinck[1,★], Oliver Elbert[1,4,★], Rhea George[1,★], Jeremy McGibbon[1,★], Tobias Wicky[1,★], Elynn Wu[1,★], Christopher Kung[2], Tal Ben-Nun[3], Lucas Harris[4], Linus Groner[5], and Oliver Fuhrer[1,6]

[1]Climate Modeling, Allen Institute for Artificial Intelligence, Seattle, WA, USA
[2]Global Modeling and Assimilation Office, Goddard Space Flight Center, NASA, Greenbelt, MD, USA
[3]Department of Computer Science, ETH Zurich, Zurich, Switzerland
[4]Weather and Climate Dynamics Division, Geophysical Fluid Dynamics Laboratory, NOAA, Princeton, NJ, USA
[5]Swiss National Supercomputing Centre (CSCS), ETH Zurich, Lugano, Switzerland
[6]Numerical Prediction, Federal Office of Meteorology and Climatology MeteoSwiss, Zurich, Switzerland
[★]These authors contributed equally to this work.

Correspondence: Oliver Fuhrer (oliver.fuhrer@meteoswiss.ch)

Received: 16 September 2022 – Discussion started: 28 September 2022
Revised: 21 December 2022 – Accepted: 9 February 2023 – Published: 17 May 2023

Abstract. Progress in leveraging current and emerging high-performance computing infrastructures using traditional weather and climate models has been slow. This has become known more broadly as the software productivity gap. With the end of Moore's law driving forward rapid specialization of hardware architectures, building simulation codes on a low-level language with hardware-specific optimizations is a significant risk. As a solution, we present Pace, an implementation of the nonhydrostatic FV3 dynamical core and GFDL cloud microphysics scheme which is entirely Python-based. In order to achieve high performance on a diverse set of hardware architectures, Pace is written using the GT4Py domain-specific language. We demonstrate that with this approach we can achieve portability and performance, while significantly improving the readability and maintainability of the code as compared to the Fortran reference implementation. We show that Pace can run at scale on leadership-class supercomputers and achieve performance speeds 3.5–4 times faster than the Fortran code on GPU-accelerated supercomputers. Furthermore, we demonstrate how a Python-based simulation code facilitates existing or enables entirely new use cases and workflows. Pace demonstrates how a high-level language can insulate us from disruptive changes, provide a more productive development environment, and facilitate the integration with new technologies such as machine learning.

1 Introduction

Current weather and climate models are written in low-level compiled languages such as Fortran for performance (Méndez et al., 2014) and typically run on high-performance computing (HPC) systems with CPUs. With the end of Moore's law (e.g., Theis and Wong, 2017), HPC systems are increasingly relying on specialized hardware architectures such as graphics processing units (GPUs) to increase throughput while maintaining a reasonable power envelope (Strohmaier et al., 2015). This has led to a number of efforts to port existing weather and climate models to run on such heterogeneous architectures, for example by adding OpenACC directives (Lapillonne et al., 2017; Clement et al., 2019; Giorgetta et al., 2022). Today, there are a handful of successful productive deployments of weather and climate models (e.g., COSMO, Fuhrer et al., 2018, and Model for Prediction Across Scales, MPAS) on GPU-accelerated supercomputers. But porting large code bases using compiler directives comes at a price: the maintenance cost increases significantly due

to more hardware-specific details being explicitly exposed in the user code and due to the fact that conditional compilation increases code complexity and the likelihood of introducing errors. Also, optimizations often need to be tailored to a hardware target, which may lead to code duplication and specialized implementations. As a result, community codes have been slow in their adoption of novel and emerging hardware architectures, and increasingly complex code bases hinder their further development. The term *software productivity gap* has been coined to describe this situation (Lawrence et al., 2018).

Alternative approaches are being explored: the Simple Cloud-Resolving E3SM Atmosphere Model (E3SM Project, 2021) is a global atmosphere model implemented in C++ using the Kokkos library (Trott et al., 2022), which provides abstractions for parallel execution and data management for a wide range of programming models and target architectures using a technique called template meta-programming. LFRic, the next-generation weather and climate modeling system being developed by the UK Met Office (Adams et al., 2019), is implemented using a domain-specific language (DSL) embedded in Fortran and leverages a domain-specific compiler named PSyclone to generate executable parallel code. While these approaches are very promising, they have currently not been adopted more widely in the field of weather and climate science.

A compelling alternative approach has been extremely successful in the trending field of machine learning: frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2015) have accelerated the rapid and broad adoption of machine learning methods. These frameworks allow users to implement algorithms with an abstract high-level implementation in Python using a syntax which is reminiscent of doing computation using the NumPy library (C. R. Harris et al., 2020). Different back ends allow users to target a diverse set of hardware architectures for efficient execution while retaining a high degree of programmer productivity. The approach has been shown to scale, and the computational effort to train such models (e.g., Brown et al., 2020) is comparable in size to high-resolution weather and climate simulations.

Aside from the model code itself, scientists developing and using compiled models have increasing needs to interface model code with scripting languages online. The drastic increase in model resolution over the past decades has increased the need for online diagnostic calculations to avoid slow I/O operations. It can also simplify development of machine learning parameterizations to be able to interface Python code with a compiled model. This has motivated scientists to interface Python code with models online, for example by calling Python from Fortran (Brenowitz and Bretherton, 2019; Partee et al., 2021a, b) and by wrapping Fortran models to be driven by Python (Monteiro et al., 2018; van den Oord et al., 2020; McGibbon et al., 2021).

We present Pace, an open-source performance-portable implementation of the FV3 dynamical core (Putman and Lin, 2007; Harris et al., 2021) and GFDL microphysics (Chen and Lin, 2013; Zhou et al., 2019) written entirely in Python. Pace uses the GridTools for Python (GT4Py) DSL, which separates the definition of numerical algorithms from the specific implementation for a given hardware architecture. Optimization details such as storage order, execution schedule, placement of data in memory hierarchy, and loop bounds are not the responsibility of the domain scientist. This allows the use of a single unified and concise code base across hardware back ends, which clearly presents numerical operators and executes efficiently. The same model code can be used for applications in the classroom inside a Jupyter Notebook as well as deployment at scale on large high-performance computing systems. Using Python as the host language enables highly productive model development, testing, and validation workflows. By having access to a large ecosystem of well-maintained Python packages, entirely new workflows are enabled (see Sect. 6.2).

The outline of the paper is as follows. We start in Sect. 2 with an overview of the DSL structure before moving on to Pace itself in Sect. 3. In Sect. 4 we discuss the process of porting and validating the code from Fortran to Python in detail. Section 5 highlights the performance of Pace, and Sect. 6 showcases important features of Pace, especially use cases and the benefits of developing in Python. Section 7 documents the limitation of Pace, and finally we summarize in Sect. 8.

2 Python-based DSL: separation of concerns

2.1 A modern DSL: requirements

The software productivity gap problem can be described as an issue with the diverse set of skills required by climate modelers to implement a production-grade climate and weather model, from discretizing the underlying physical equations all the way to the optimization details of a given hardware architecture. This process can be roughly split in two:

1. climate and weather modeling – scientific motivation, algorithmic design, and numerical implementation;

2. performance development – adapting the implementation to a set of given hardware architecture and optimization to reach a useful simulation time.

There is overlap between these; some algorithmic designs are better suited to certain hardware architectures, and implementation details can be changed to improve model speed, but this breakdown is a useful heuristic for the development of weather and climate models.

The design of a modern domain-specific language needs to respond to both classes of users. For climate modelers a DSL should

- be easy to use, complete with debugging tools and simple methods to extracting scientific results;
- allow easy ways to implement new features and or ways to escape the DSL as new methods are developed;
- enable quick development round-trip;
- run with optimal performance;
- improve development by simplifying the implementation of common code patterns.

For the performance developers it should

- leverage a proven host language to strengthen basic development of a compiler tool chain and work on solid ground;
- build a maintainable and extensible set of compiler elements in order to keep up with novel and emerging hardware architectures;
- ensure the presence of a lower-level interface to perform generic or custom optimizations.

Python stands as a good candidate language to host a DSL given its robustness, ease of use, wide adoption across a range of research and industry groups, large ecosystem of pre-existing tools and packages, and capacity for introspection as an interpreted language. Furthermore, in the weather and climate community Python has established itself as the lingua franca for analysis, visualization, and post-processing. But Python has serious limitations in terms of execution performance: designed primarily as a scripting language, Python alone cannot achieve the performance required by workloads running on large HPC systems.

The solution is to escape Python at runtime and leverage compiled code. This is a technique already heavily used by frameworks such as Cython (Behnel et al., 2011) and Numba (Lam et al., 2015) (see, e.g., Augier et al., 2021). The DSL compiler is responsible for translating and compiling (transpiling) code written in Python into another, more performance-oriented language. The generated code can be tailored and optimized to a specified hardware target via hardware-specific compiler back ends.

2.2 Related work

For this work we use the GT4Py DSL compiler because the DSL provides a single-source solution for portable performance with the correct abstractions for weather and climate models. There are, however, other approaches using Python at different levels of abstraction that are worth discussing.

Cython (Behnel et al., 2011) is an optimizing static compiler based on Python and the Pyrex language. It is a powerful tool for accelerating Python code, but it lacks the portability and high-level abstractions we seek. Other packages provide portability but still lack the domain-level abstractions to simplify the language for climate scientists. An example of such a package is Numba (Lam et al., 2015), which is a just-in-time (JIT) compiler for Python and NumPy code. JAX (Bradbury et al., 2018) similarly lacks the weather and climate abstractions we desire in the front-end DSL language but brings with it convenient features such as adjoint capability. Other application-building frameworks such as Exasim (Vila-Pérez et al., 2022), FEniCS (Alnaes et al., 2015), and Dedalus (Burns et al., 2020) mostly operate on the partial-differential-equation (PDE) level, which allows for more automated model development at the expense of flexibility in the discretization. In order to faithfully reproduce all aspects of the FV3 dynamical core and GFDL cloud microphysics, we required greater flexibility than these frameworks provided. Thus, the DSL we are targeting works at a lower level on the mathematical representation after the PDE has been discretized.

2.3 GT4Py: a Python-based DSL

Our implementation of the domain-specific language described above is called GridTools for Python, or GT4Py. Developed in partnership with the Swiss National Supercomputing Centre (CSCS), it defines a DSL on top of Python. The code is analyzed and compiled into a C++ or CUDA executable that is bound back to the original Python, creating a seamless experience for the modelers but enabling fast and optimized execution (Fig. 1). On top of those performance back ends, GT4Py also provides a back end to generate NumPy code, which is useful for debugging and quick development as no compilation is required. One GT4Py back end leverages the DaCe (Ben-Nun et al., 2019) framework to be able to also include regular Python code in between DSL code in the translation.

GT4Py operates through the use of stencils: inside a GT4Py stencil each grid cell in an array is modified according to neighboring cells based on a fixed pattern. In numerical modeling of weather and climate two major computational patterns emerge due to the reliance on three-dimensional structured or unstructured grids: computations with dependencies on the local horizontal neighborhood of a grid cell and vertical solvers with column dependencies as shown in Fig. 2. These two stencil computational patterns therefore form the basis of GT4Py's design.

GT4Py stencils treat the horizontal dimensions differently from the vertical: they always execute in parallel over the entire horizontal domain. In the vertical dimension the order of execution may be specified with the `computation` keyword, and the vertical range is set with `interval`. A horizontal stencil can be written `with`

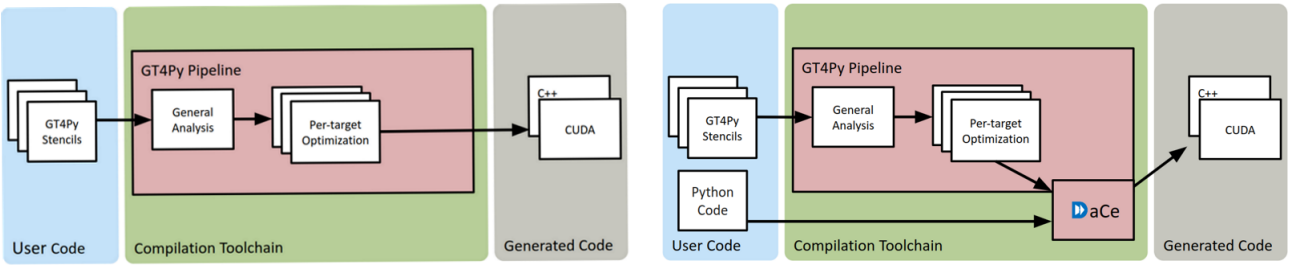

Figure 1. Workflow pipeline of GT4Py (left) and GT4Py combined with DaCe (right). User code is analyzed, optimized, and translated into hardware-target-specific optimized code. In the case of DaCe, the Python code in between GT4Py stencils (control flow) is also included in the translation. This can lead to significant performance improvements.

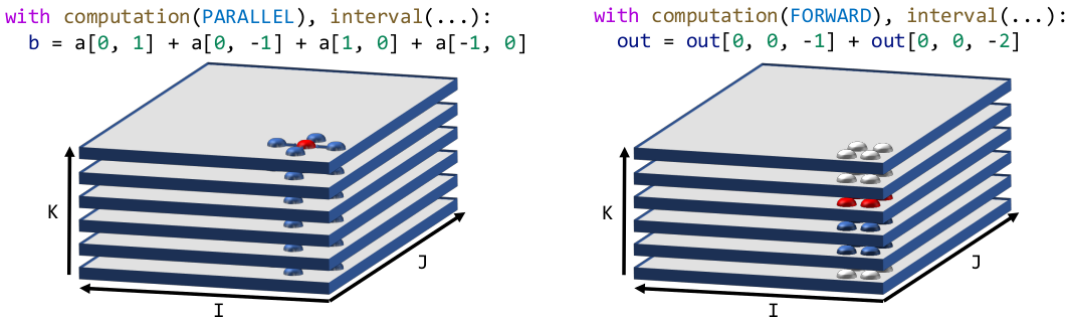

Figure 2. Two typical computational patterns in weather and climate models: on the left is a horizontal stencil, and on the right is a vertical solver. Figure from Ben-Nun et al. (2022).

`computation(PARALLEL)` to parallelize over the vertical domain as well if there are no vertical loop-carried dependencies. A column-based stencil that calculates up or down the k dimension, on the other hand, can be implemented sequentially in k `with computation(FORWARD)` or `BACKWARD`, respectively. Each stencil can also have multiple `computation` and `interval` blocks for flow control. Because stencils are applied to each point in a 3D grid, all indexation within a stencil is relative to the current computed grid point; i.e., `Array[0,0,0] - Array[1,0,0]` takes each array element and subtracts the array element immediately to its right along the x axis. This method of indexing three-dimensional arrays allows the modelers to use the same indexing convention (I, J, K), irrespective of the actual storage layout in memory.

GT4Py also allows zero-cost function calls, enabling more readable and reusable code within models. Extents of the computation are automatically determined by the DSL, including through these function calls. Figure 1 shows an example stencil function that executes in parallel over the entire vertical dimension. The domain of dependence for intermediate variables (`advected_u` and `advected_v`) is automatically determined by GT4Py. In this example, the advection helper functions take a horizontal difference of the `u` and `v` contravariant velocity components `ub_contra` and `vb_contra`, respectively. As a result, the interpolation function which computes these values is applied on a larger domain than the final operation in the stencil.

In order to implement the FV3 dynamical core we had to extend the stencil concept multiple times, enabling four- and two-dimensional arrays, loops over the k axis, and specialized handling of horizontal subdomains. Because GT4Py is written in Python, those extensions were easy to develop and they will be discussed further in Sects. 4 and 5.

GT4Py is able to optimize the performance of stencils; however, some Python overhead remains in the code linking the stencil operations together. In order to resolve this and optimize the entire Pace model we have incorporated DaCe (Ben-Nun et al., 2019) as a back end for Pace. Using this framework both GT4Py stencils and raw Python code are exposed to optimization and transpilation to high-performance C++ or CUDA executables, removing all Python overhead seamlessly for the modelers. Section 5 contains more information on code optimizations, while Ben-Nun et al. (2022) details the performance optimizations more thoroughly.

3 Pace

Pace is a GT4Py implementation of the nonhydrostatic FV3 dynamical core and GFDL microphysics (Putman and Lin, 2007; Harris et al., 2021; Chen and Lin, 2013; Zhou et al., 2019). It is based on the same version of the National

```
def compute_kinetic_energy(
    vc: FloatField,
    uc: FloatField,
    cosa: FloatFieldIJ,
    rsina: FloatFieldIJ,
    v: FloatField,
    vc_contra: FloatField,
    u: FloatField,
    uc_contra: FloatField,
    dx: FloatFieldIJ,
    dxa: FloatFieldIJ,
    rdx: FloatFieldIJ,
    dy: FloatFieldIJ,
    dya: FloatFieldIJ,
    rdy: FloatFieldIJ,
    dt_kinetic_energy_on_cell_corners: FloatField,
    dt: float,
):
    with computation(PARALLEL), interval(...):
        ub_contra, vb_contra = interpolate_uc_vc_to_cell_corners(
            uc, vc, cosa, rsina, uc_contra, vc_contra
        )
        advected_v = advect_v_along_y(v, vb_contra, rdy=rdy, dy=dy, dya=dya, dt=dt)
        advected_u = advect_u_along_x(u, ub_contra, rdx=rdx, dx=dx, dxa=dxa, dt=dt)
        dt_kinetic_energy_on_cell_corners = (
            0.5 * dt * (ub_contra * advected_u + vb_contra * advected_v)
        )
        dt_kinetic_energy_on_cell_corners = all_corners_ke(
            dt_kinetic_energy_on_cell_corners, u, v, uc_contra, vc_contra, dt
        )
```

Listing 1. An example stencil definition function that computes the kinetic energy on cell corners. The functions `advect_v_along_y`, `advect_u_along_x`, and `all_corners_ke` are defined outside the stencil and the DSL compiler inlines the relevant code. FloatField and FloatFieldIJ are GT4Py-specific data types for three-dimensional and two-dimensional fields, respectively. FloatField is a type which is used to declare three-dimensional fields of configurable floating-point precision.

Oceanic and Atmospheric Administration (NOAA) Unified Forecast System (UFS) model as McGibbon et al. (2021), forked from the UFS repository v1 in December 2019 (Zhou et al., 2019), and is nearly identical to the dynamical core used in SHiELD (L. Harris et al., 2020). At present Pace only supports nonhydrostatic, uniform-resolution simulations, with a restricted set of subgrid reconstruction schemes (`hord` and `kord` values). Pace has the ability to read initial conditions generated by the Fortran model and other saved outputs and can also generate initial conditions for analytic test cases. Currently Pace supports six-tile gnomonic cubed-sphere grids and single-tile orthogonal, doubly periodic grids, though only at uniform resolution. Future development will enable nested and stretched grids as described in Harris and Lin (2013) and will integrate the rest of the physics parameterizations (Zhou et al., 2019). Pace is MPI-enabled, allowing it to run in parallel, but can also run using a serial communicator, running each rank in serial and saving data files to mock MPI communication.

3.1 A modular model

Pace is designed to be modular; each model component of Pace (e.g., dynamics, microphysics, utilities, DSL integration) exists as a separate package. Computationally focused packages like the dynamics contain hierarchies of component modules (Fig. 3). These components provide clear boundaries to document and change model behavior. For example, the horizontal transport scheme used by FV3 (Putman and Lin, 2007; Lin and Rood, 1996) takes any one-dimensional finite-volume subgrid reconstruction scheme satisfying certain numerical conditions and can extend it to two dimensions. Within Pace, one-dimensional subgrid reconstruction code is contained in the XPPM and YPPM modules (X and Y piecewise parabolic methods, respectively). These modules take in scalar grid-cell-mean values and Courant numbers (speed as a fraction of grid-cell width) defined on transport interfaces and return the average value of the scalar within the section of grid cell to be advected through the cell interface. The FiniteVolumeTransport class extends these one-dimensional subgrid reconstruction schemes to produce two-dimensional horizontal fluxes. This allows a scientist to modify the behavior of the dynamics by replacing only the XPPM and YPPM components, for example with a cubic reconstruction scheme or a subgrid reconstruction scheme based on machine learning.

To maintain simplicity of the code and facilitate the separation of compile time and runtime, Pace uses a simple object-oriented framework that expresses nearly all of the internal computational components as initializable functions, as shown in Listing 2. Each of these functions is defined as a Python class with an initialization method and a call method. Initialization performs the necessary setup including memory allocation and compilation of DSL code. Each component's initialization code defines the component's temporary variables, compiles or loads its stencils, and recursively

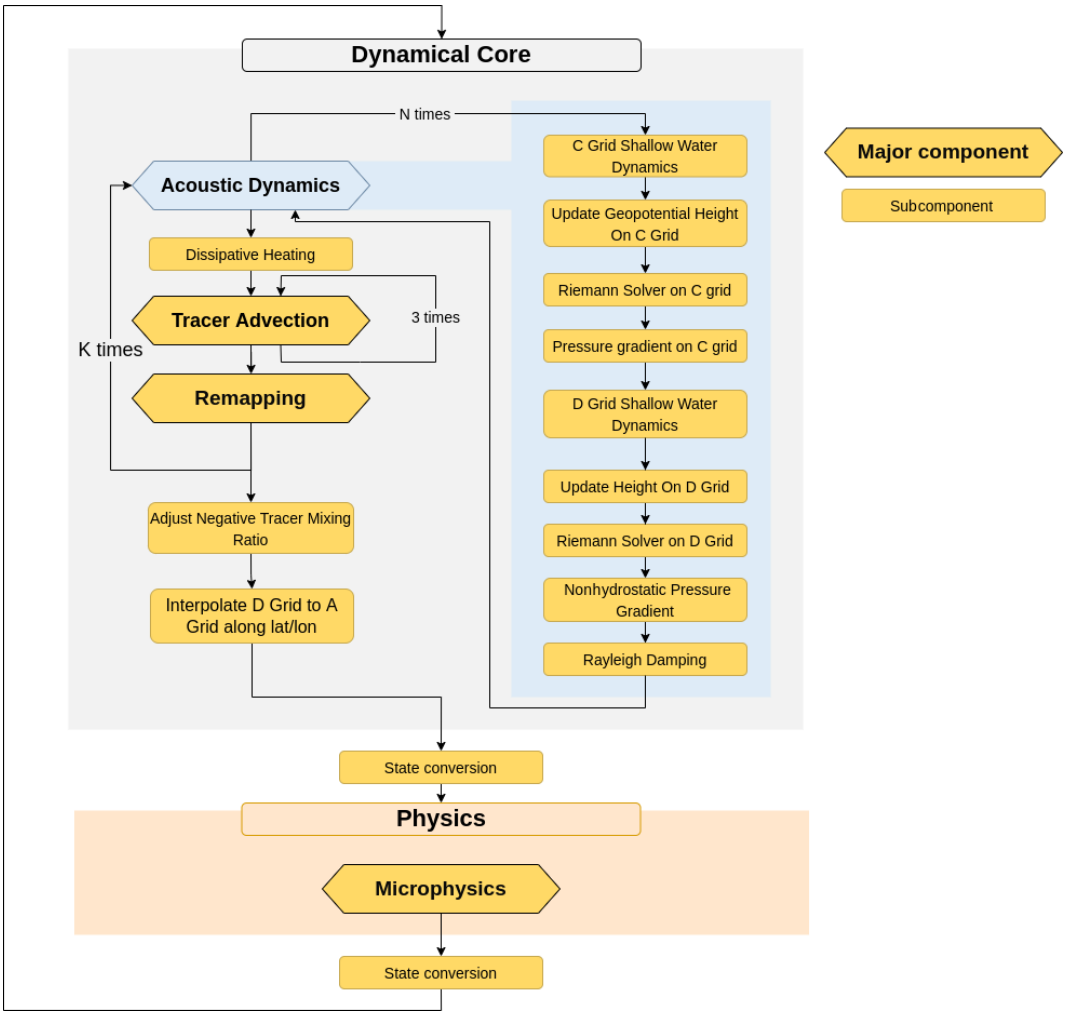

Figure 3. Internal structure of the Pace dynamical core. To reduce instabilities from the Lagrangian vertical coordinate the acoustic dynamics, tracer advection, and vertical remapping are sub-stepped, executing K times during a model time step. The acoustic dynamics are additionally sub-stepped to ensure numerical stability with respect to sound waves.

initializes subcomponents. Once initialized, the component may be called the same as any other Python function. This is similar to Fortran model structures where computational code is paired with initialize and finalize subroutines.

Stencils are compiled when components are initialized, using only explicitly passed configuration data. Pace uses the factory pattern (through the `stencil_factory` argument) to reduce compilation-specific logic as much as possible within computational components. The stencil factory class implements the code responsible for allocating and compiling stencils, allowing model code to instead focus on computational motifs. In lines 8–18 of Listing 2 we can see this factory at work compiling the `compute_kinetic_energy` stencil shown in Fig. 1. The `from_dims_halo` method takes a stencil function and a set of dimensions (either cell centers or cell interfaces) to execute over and returns a compiled stencil that writes its outputs over the compute domain, in this case on cell corners (x and y interfaces). Output can be extended into the computational halos with an optional `compute_halos` argument indicating how many halo points to write (not shown). We also see how compile-time constants are passed as `externals` to the compilation method. These constants can be set by the model configuration, extracted from the domain decomposition, or passed as arguments to the initialization method, and they have the benefit of being treated as compile-time constants. Because configuration settings such as domain-decomposition and namelist settings are now known at compile time, they also provide further avenues for the DSL back end to optimize performance.

The factory pattern used here is incredibly powerful when debugging the model. For example, if a model developer finds that at some point a variable `air_pressure` used by many routines has gone negative in the model, that developer

```
class ComputeKineticEnergy:
    def __init__(self,
        stencil_factory: StencilFactory,
        grid_data: GridData,
        config: DGridShallowWaterLagrangianDynamicsConfig,
    ):
        self.grid_data = grid_data
        self._compute_kinetic_energy = stencil_factory.from_dims_halo(
            func=compute_kinetic_energy,
            compute_dims=[X_INTERFACE_DIM, Y_INTERFACE_DIM, Z_DIM],
            externals={
                "iord": config.hord_mt,
                "jord": config.hord_mt,
                "mord": config.hord_mt,
                "xt_minmax": False,
                "yt_minmax": False,
            },
        )

    def __call__(self, vc, uc, v, vc_contra, u, uc_contra,
    dt_kinetic_energy_on_cell_corners, dt):
        self._compute_kinetic_energy(
            vc=vc,
            uc=uc,
            cosa=self.grid_data.cosa,
            rsina=self.grid_data.rsina,
            v=v,
            vc_contra=vc_contra,
            u=u,
            uc_contra=uc_contra,
            dx=self.grid_data.dx,
            dxa=self.grid_data.dxa,
            rdx=self.grid_data.rdx,
            dy=self.grid_data.dy,
            dya=self.grid_data.dya,
            rdy=self.grid_data.rdy,
            dt_kinetic_energy_on_cell_corners=dt_kinetic_energy_on_cell_corners,
            dt=dt,
        )
```

Listing 2. A class that compiles and runs the stencil defined in Fig. 1. The `__init__` method initializes the `compute_kinetic_energy` stencil using the stencil definition function defined in Fig. 1, an output domain, and the constants used by the stencil, and the `__call__` method executes the stencil when the resulting object is invoked at runtime.

can temporarily insert code to immediately raise an exception if *any* stencil writes a negative value to a variable named `air_pressure`. Or that model developer could alter every stencil to write its inputs and outputs to a netCDF file when executed, while only having to modify the code used for the stencil factory.

3.2 Powerful testing

The introspection power of Python is used to great effect in the testing code for Pace. For example, we would like to ensure that Python array allocation only happens when initializing our model and not at all when it is called. To do this, we write a test which initializes the model and then replaces the storage allocation routines of GT4Py with routines which raise an exception if called before finally calling the model. If any arrays were allocated at call time, an exception would be raised and the test would fail.

We also want to ensure that the Pace model components are not stateful. The dynamical core, for example, has many temporary storage arrays assigned to it whose initial value should not matter when calling the model. However, a user could easily fail to initialize an array with zeros when they mean to, causing a bug in the model. We have a test which calls the dynamical core with the same state either once or twice and compares the value of all temporary data in the model between these two cases. It does this by dynamically crawling the Python object structure and comparing all array data. If any data differ between those two cases, it not only tells us that there is a bug but also exactly which temporary field has a bug – the one which is accessed first out of the ones which differ. Without such a test, it could take days, weeks, or months for a scientist to find the source of such an error, assuming they notice the presence of the bug.

4 Porting the model

Atmospheric models are large computational codes, making it difficult to determine the source of a bug given errors in model outputs. In order to port FV3 and the physics parameterizations we first segment the Fortran code into smaller units of code which can be ported and tested independently. Typically each unit encompasses a particular Fortran subroutine, but larger or more complicated subroutines may be broken down further to ease validation. We use the Serialbox

library to extract the inputs and outputs from each of these Fortran units. We place Serialbox compiler directives before each unit of code to serialize the inputs to that unit, and we similarly insert directives after each unit to extract its outputs. For a given test case we can then run the Fortran model to generate test data for that case and model configuration.

GT4Py stencils calculate over 3D volumes, so in our porting process we initially wrote each individual stencil to replicate a Fortran "do loop" over the i, j, and k spatial dimensions. A ported unit of code may use multiple stencils if there are computations over different horizontal domains or to increase readability of the code. In order to minimize GPU kernel launches we subsequently merged stencils that executed over the same spatial extents where possible.

4.1 Extending the DSL

In the process of writing Pace we needed to extend GT4Py in order to express the Fortran code in the DSL. FV3 discretizes the Earth into a gnomonic cubed sphere (Putman and Lin, 2007). GT4Py could only apply the same operations uniformly across the horizontal domain of a stencil. We added the ability to execute code on stencil subdomains to the GT4Py DSL language in order to perform special handling required near the corners and edges of the cubed-sphere tiles. Specifically, we included `horizontal` and `region` as stencil control keywords (similar to `computation` and `interval`) as shown in Fig. 5. `with horizontal` specifies a horizontally restricted block of code, and `region` specifies the extent of that subdomain through NumPy-like array slicing in the I and J dimensions. Though it does violate the GridTools concept of a stencil, this allows us to fully port the dynamical core and combine stencils to reproduce a natural amount of Fortran code. This also has the benefit of increasing the readability of the code: in Fortran the corner and edge computations are handled by the common pattern of if-conditionals inside of nested "for loops", while in Pace the horizontal regions are used almost exclusively for this purpose.

Another example is in the vertical remapping component of the model. The FV3 dynamical core uses a Lagrangian vertical coordinate which is regularly remapped to its original Eulerian grid using the piecewise parabolic method (Lin, 2004). The remapping process requires a double k loop over the vertical dimension: an inner loop over the deformed vertical levels to sum their contributions to an Eulerian level and an outer loop over those Eulerian levels. When we initially ported the remapping code GT4Py did not support "for loops" or "while loops" inside of stencils, so we wrote the inner loop as a stencil over the deformed k levels and the outer loop in plain Python. While this implementation worked algorithmically, calling a stencil inside of a "for loop" has a large penalty in model speed due to the repeated kernel launches and the removal of the loop structure from the DSL optimization path. To resolve this we added "while loops" to GT4Py stencils. This allows us to consolidate the remapping code into one stencil over the Eulerian k levels, remove the Python "for loop", and expose the entire remapping scheme to the DSL compiler. This reduced the runtime of the remapping step by over an order of magnitude.

GT4Py is able to cover a large number of algorithmic motifs used in weather and climate models. This section illustrated two examples where the DSL has been extended in order to express motifs which were present in FV3GFS. There is also an ongoing effort to extend GT4Py to unstructured grid computations. Nevertheless, any DSL will naturally be restricted from covering some algorithmic motifs. For GT4Py these include reductions (e.g., computing horizonal integrals over a field), interpolation (e.g., for multigrid), and search patterns (e.g., looking for a neighboring grid cell with certain properties). Whether these patterns should fall into the scope of the GT4Py DSL or another framework is a design decision, which can be weighed for example against compiling these patterns with DaCe.

4.2 Model validation

Each unit of code is validated by running it with the input data serialized from runs of the Fortran model, and comparing the outputs of our code to the serialized Fortran outputs. If a ported unit is particularly large (e.g., vertical remapping, grid generation) we serialize data from components of the larger unit and test the equivalent Pace components to reduce the complexity of each test. We use two sets of initial atmospheric conditions as our test cases, each with a corresponding configuration namelist. Our "standard" test case is generated from NCEP reanalysis data from 00:00:00Z on 1 August 2016 (as in McGibbon et al., 2021), and the other is the baroclinic instability test case described in Jablonowski and Williamson (2006). Initial versions of the DSL code were tested on the standard case run on six MPI ranks (one per tile) and a 12 by 12 horizontal grid on each tile face with 79 vertical levels. We also test with a 54-rank domain decomposition, as this gives each tile face a rank for each corner and edge as well as a central rank with no edges. Our tests thus cover the range of edge and corner handling required of a given rank from no corners or edges to all eight corners and edges.

In many cases the Pace outputs can be brought near roundoff error to the Fortran outputs. However, there are some instances where changes to the order of operations or the use of transcendental functions makes such reproducibility impossible, so we must choose a validation threshold for our code. We do this by perturbing the inputs to the Fortran code by small, floating-point differences and comparing the outputs. This leads us to adopt relative differences of 10^{-14} as our default validation threshold. If we find validation errors greater than our default, we test the components of the offending code to see what operations introduce discrepancies between Pace and Fortran. Often these differences are due to errors in-

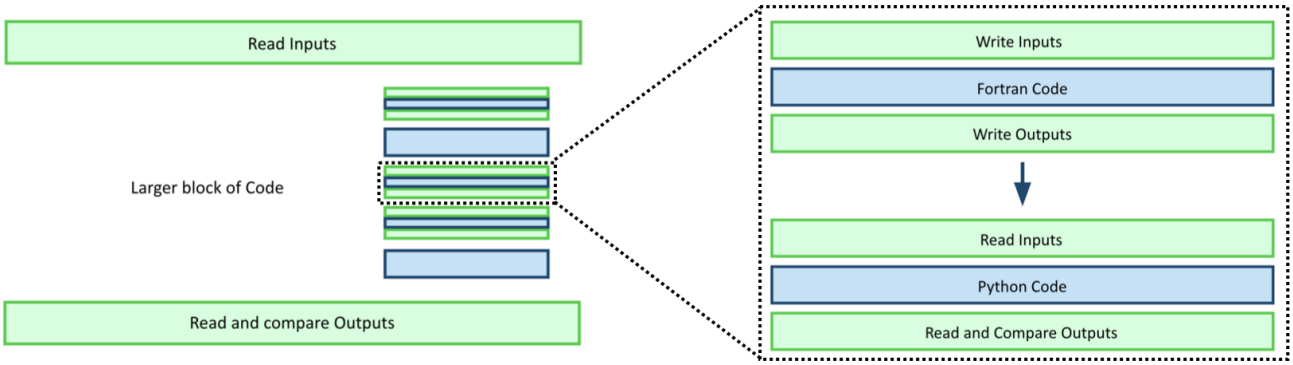

Figure 4. Schematic of our porting strategy: we port small units of code, test that they validate against the original Fortran, and then assemble them into larger model components we then validate, building up to a full model port.

```
@gtscript.stencil(...)
def set_field(field: Field[float]):
    with computation(PARALLEL), interval(...):
        field = 0.0
        with horizontal(region[i_start, :]):
            field = 1.0
        with horizontal(region[i_start + 1, :]):
            field = 2.0
        with horizontal(region[i_start + 2, :]):
            field = 3.0
```

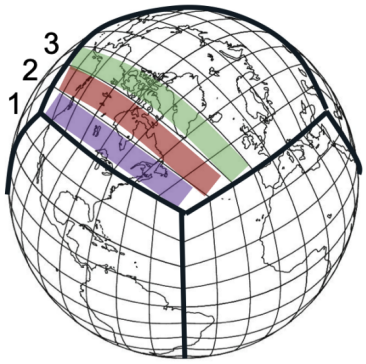

Figure 5. Example illustration of how horizontal regions in GT4Py can be used to specialize computations in certain sections of the computational domain: the first region assigns 1.0 to the first row of the array, the second region assigns 2.0 to the second row, and the third assigns 3.0 to the third row.

troduced during porting and we can reconcile Pace with the Fortran code. If these differences are instead due to algorithmic changes (reordering, etc.) we set the error tolerance for that piece of code based on the underlying Fortran–Python difference. For example, in the vertical remapping scheme FV3 makes use of multiple `goto` statements for its control flow. These are not available in GT4Py, and so some code has been rearranged to replicate the original algorithm, introducing small deviations from the Fortran outputs.

When we are confident that each component of the model accurately reproduces the Fortran version, we test larger combinations of these units to ensure the implementation of these larger modules also matches Fortran, as illustrated in Fig. 4. For example, after validating the cubic-spline vertical interpolation code and the code that calculates the contributions from deformed, Lagrangian pressure levels to the remapped Eulerian levels, we then validate the code to vertically remap a single variable from Lagrangian to Eulerian pressure levels. When this code validates, along with the tracer remapping, saturation adjustment, and moist potential temperature adjustment code, we can then validate the entire vertical remapping scheme. In this way we hierarchically assemble and validate Pace against the Fortran code. These validation tests are incorporated into our suite of continuous integration tests to ensure that future developments and code changes do not affect model validity.

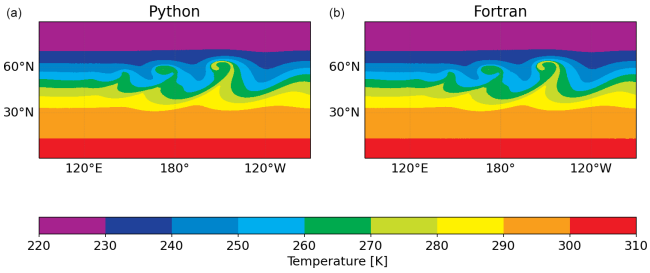

Figure 6. Detail of 850 mbar temperature of the baroclinic instability simulated with the Pace **(a)** and Fortran **(b)** dynamical cores on day 9 at 10 km resolution. These results can be compared to Fig. 6 of Jablonowski and Williamson (2006) and show how well our model replicates the original Fortran.

Once the full Pace dynamical core passes these unit tests, we run the Jablonowski and Williamson (2006) baroclinic instability test case for 9 model days and compare the results against the same test run in Fortran. In this test case, the dynamical core is initialized with zonally symmetric steady-state winds, on top of which a Gaussian perturbation of the zonal wind is superimposed in the northern mid-latitudes, triggering the evolution of a baroclinic wave. In Fig. 6 we show the region of interest in a 10 km resolution simulation ($960 \times 960 \times 80$ grid cells per tile, run on 150 MPI ranks) of the same instability from Pace and the reference Fortran model. We see that the model behavior is consistent with the reference Fortran model and replicates the results from the highest horizontal resolution in Jablonowski and Williamson (2006) well. Figure 7 shows the difference in 850 mbar temperature between Pace and Fortran running the test case at 200 km ($48 \times 48 \times 80$ grid cells per tile) resolution. Both models were run with six MPI ranks. The Pace dycore matches the Fortran code closely early on, with random errors on the order of 10^{-11} after 3 model days of integration. Due to the nature of the baroclinic instability these small errors do eventually grow to the order of 10^{-1} after 9 simulation days (3888 time steps), but even on day 6 the errors are only on the order of 10^{-5}. Based on these results we are confident that Pace accurately reproduces the Fortran model code.

Pace has only been validated using double-precision floating-point arithmetic and values. All performance results shown in the next section have been measured using double precision.

5 Model performance

All experiments were conducted on the Piz Daint supercomputer at CSCS. Piz Daint contains 5704 Cray XC50 nodes, with an Intel Xeon E5-2690 v3 12-core CPU, one NVIDIA Tesla P100 GPU (16 GB RAM), and 64 GB of host RAM on each node. The nodes are connected via the Cray Aries interconnect. We are using Python 3.8.2 with Pace revision 0.2. The generated code was compiled with CUDA 11.2 and GCC 9.3.0. To expose the full node architecture to the DSL optimization scheme we run Pace with one MPI rank per node for both CPU and GPU back ends. The reference Fortran model was compiled with Intel IFORT version 19.1.3.304. We run the Fortran code in its optimal configuration of six ranks per node and four threads per rank, fully utilizing all 24 virtual cores available with hyperthreading.

Table 1 shows our model performance and strong scaling results, comparing the absolute runtime of the Pace dynamical core on both CPU and GPU architectures with the original Fortran implementation. Figure 9 presents these data graphically. We show our weak-scaling results in Fig. 8. The microphysics implementation in Pace takes a negligible amount of time ($\lesssim 1\,\%$), so we focus our discussion on the dynamical core.

Our weak-scaling results (Fig. 8) demonstrate a speedup of $3.6\times$ against Fortran that is nearly constant across simulation scales. The slight decrease in runtime at higher scales is due to the heavier computational load on the limited case of six MPI ranks; when running on six MPI ranks (one rank per tile face) FV3's corner and edge handling has to be computed on every rank, while at higher scales that computational load is better spread across MPI ranks. Section 5.3 discusses this in more detail. Overall we see that Pace exhibits perfect weak scaling, which validates the capacity of a GPU-running model to simulate at kilometer-scale resolutions in an efficient way.

Since we observe perfect weak scalability, meaning an increase in compute nodes working on a problem does not affect the total time, we can determine that the total time only scales with the work per node. Since the work is increasing linearly with the number of grid points per node, we show a detailed analysis of the scalability of Pace with varying work per node.

The domain size scaling experiments shown in Fig. 9 show increasing performance gains on GPU as larger domains are simulated on each compute node. This is expected, as GPUs are well fitted to high domain sizes since they are capable of higher throughput than CPUs when optimized accordingly. As is well known and described for example in Fuhrer et al. (2018), offloading computation to GPUs comes with a non-insignificant startup cost that only starts to pay off if enough work is done on accelerators. Pace is no exception to this rule. We can see a regime where not enough work is done to justify the startup cost.

A detailed view of the scaling numbers for the most relevant region is summarized in Table 1. When we maximize the amount of data on each node – shortly before we run out of memory on the GPU – Pace achieves a speedup of $3.2\times$ in comparison to Fortran. The CPU version, on the other hand, is $0.2\times$ the speed of the Fortran code. This is due to a combination of two factors:

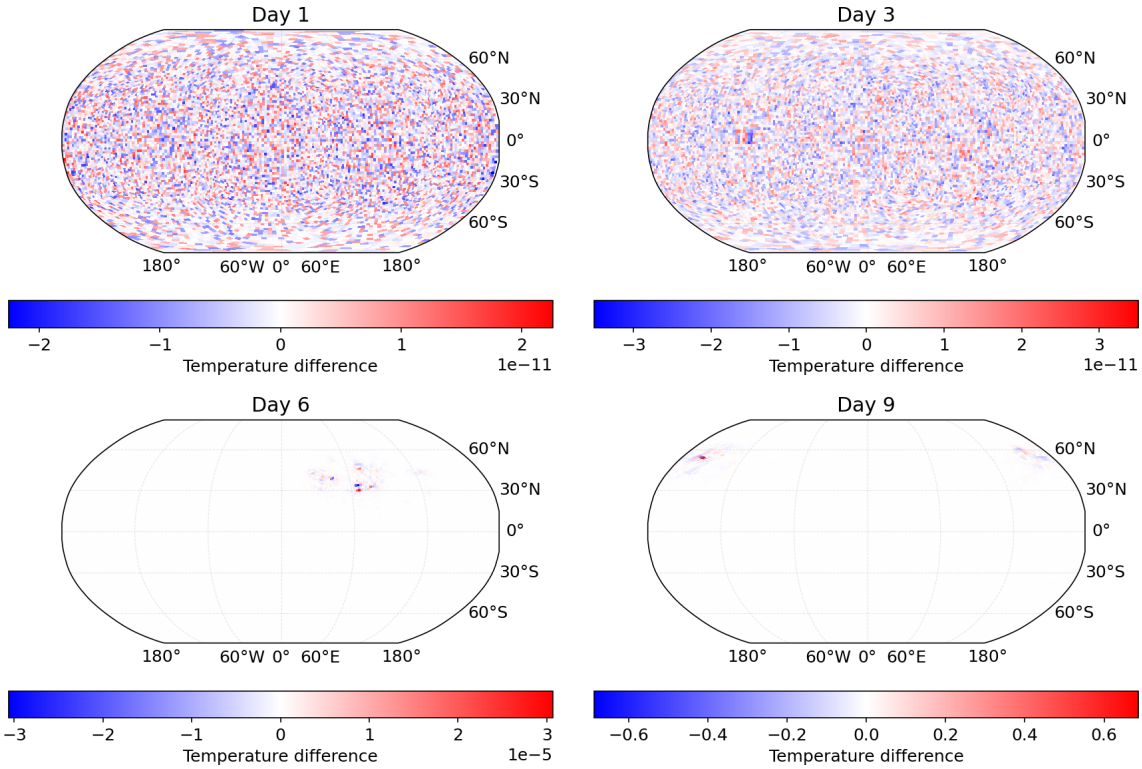

Figure 7. The 850 mbar temperature difference between Pace and baseline Fortran simulations of a baroclinic instability at 200 km resolution after 1, 3, 6, and 9 model days, showing a good match between the two models.

Table 1. Performance metrics comparing the Pace dynamical core with the Fortran reference code. The size (in number of grid points) of each face (tile) of the cubed sphere grid is increased from row to row. Each of the configurations is run on six compute nodes, one compute node per face of the cubed-sphere grid. Essentially this corresponds to global simulations of decreasing grid spacing of 96, 72, and 48 km, respectively. The time measurements are the execution time of one invocation of the dynamical core (see Fig. 3).

Tile size	Scaling	Fortran		Pace CPU			Pace GPU		
		Time [s]	Scaling	Time [s]	Scaling	Speedup	Time [s]	Scaling	Speedup
$108 \times 108 \times 80$	–	3.58	–	16.00	–	0.22	1.98	–	1.81
$128 \times 128 \times 80$	$\times 1.40$	4.66	1.30	22.25	1.39	0.21	2.34	1.18	1.99
$192 \times 192 \times 80$	$\times 2.25$	12.74	3.56	48.07	3.00	0.27	3.98	2.01	3.20

- the Fortran model we ported is itself highly optimized for CPU;
- the optimization effort was geared toward demonstrating the viability of the Python DSL for GPU usage.

Despite focusing on GPU optimization, some of the optimization methods also improved CPU performance, and the functional CPU back end does demonstrate portability. While outside the scope of this paper, work to improve CPU performance is now underway.

Model performance is a core motivation for GPU acceleration and for adopting a DSL, and it is encouraging to see a significant speedup between Pace GPU and Fortran versions. While the CPU performance has not been our focus to date, it is the next priority for Pace development. We discuss our optimization strategies and explain how model performance influenced our decision process in porting the model in the following subsections. More detailed performance results and a thorough analysis are available in Ben-Nun et al. (2022). The version of the model code and supporting framework for Ben-Nun et al. (2022) is slightly earlier, hence a slight difference in absolute numbers, but the methodology remains the same.

5.1 Optimizations

Because GT4Py has multiple back ends for various target architectures there is no one performance number that captures the entirety of our approach. Nonetheless, the aggregate per-

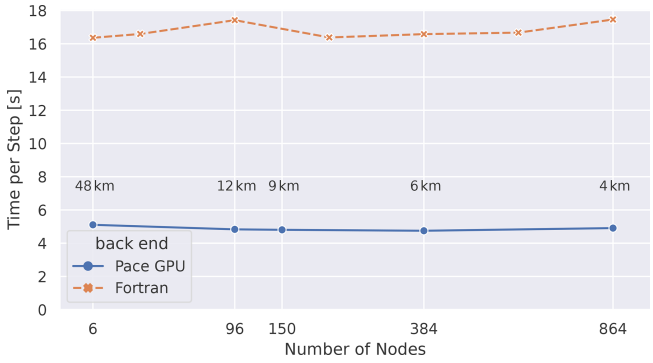

Figure 8. Weak-scaling analysis from Ben-Nun et al. (2022): execution time for one invocation of the dynamical core (see Fig. 3) is shown as a function of an increasing number of compute nodes (each with one GPU). In this setup Pace is run with one MPI rank per node, while the Fortran code is run with six MPI ranks per node and four threads per rank. In this weak-scaling experiment, the number of grid points per GPU are kept constant at $192 \times 192 \times 79$. As more nodes are added, the resolution of these global simulations increases. The average grid spacing is indicated in text, and a maximum of 4 km is achieved when running on 864 nodes.

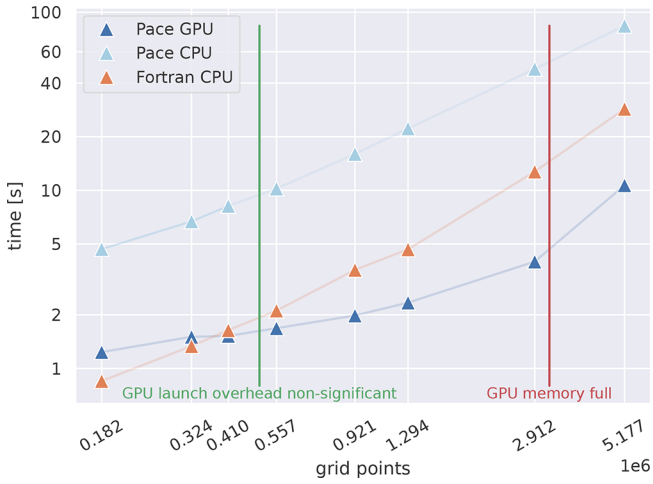

Figure 9. Dynamical core runtime for varying domain sizes, expanded from Table 1. The time measurements indicate the execution time of one invocation of the dynamical core (see Fig. 3).

formance across back ends indicates the power of the DSL paradigm through performance portability. With the capability of code generating for specific hardware targets, we are not limited to single, catch-all solutions in our user-facing code. Instead we can generate optimized code for each type of hardware individually through back-end logic, allowing user code to focus on numerical details. As an example, on CPU architectures it usually pays off to use coarse-grained parallelism such that threads are assigned to different subdomains, while on GPUs the parallelization strategy involves having blocks of threads execute subdomains.

Weather and climate models are written with large configuration files, namelists, to allow for flexible use. In order to support all the possible configurations, standard models use many conditionals that can not be resolved at compile time, such as which subgrid reconstruction to use for tracer advection. This limits how aggressively the low-level compiler can optimize the model. Code generation from a DSL allows us to generate code and compile the model for a specific namelist configuration to circumvent this problem.

Furthermore the DSL compiler is able to apply domain-specific optimizations under certain conditions on a per-stencil level before generating code at all. These include the following.

1. *Reduce main memory accesses.* Replace 3D temporary fields used for intermediate results that are stored in global memory with smaller buffers that allow for more reuse and faster access.

2. *Inlining.* Fully inline both function calls as well as nested conditionals, replacing function calls and branching conditionals with the relevant code at compile time and removing the performance overhead of these code patterns.

3. *Pruning.* Analyze the code based on all the compile-time constants provided and prune unreachable branches of code.

4. *Fusion.* Fuse numerical operators defined in separate functions and contexts into single kernel calls, as long as that does not create race conditions. This optimizes performance by reducing the amount of synchronization needed as well as giving the underlying general purpose compiler more flexibility.

5.2 DaCe

In order to achieve good performance in a Python-driven environment it is crucial to minimize the overhead moving from the driver language to the compiled language executable. Since our implementation of the model is a series of compiled stencils called from Python, the performance overhead linked to calling the compiled stencils from Python scales with the number of stencils called. A secondary issue is that with the fragmentation into individual stencils we limit the optimization potential of the DSL, as it is limited to the scope of a single stencil for the optimizations described above.

We leverage DaCe (Ben-Nun et al., 2019) to address those shortcomings, providing a full-program optimization framework. With our DaCe back end in GT4Py we are able to compile the entire loop over time steps into a single executable called from Python, thus completely removing all Python to C call overhead during our simulation. With DaCe as our back end we are able to leverage custom optimization for our code, including improving the computational layout, improving where and how memory allocation happens,

scheduling computation to improve parallelism as well as increasing data locality, and improving the pressure on global memory. A detailed explanation of our approach and how it affected performance can be found in Ben-Nun et al. (2022).

5.3 Corner and edge handling

As discussed in Sect. 4.1, the cubed-sphere discretization of FV3 (Putman and Lin, 2007) requires special finite-difference stencils applied near edges and corners to account for the grid geometry. We added the ability to execute stencils on horizontal subdomains to enable these motifs in GT4Py, which unfortunately has negative performance implications for Pace.

FV3 parallelized the vertical plane K instead of the horizontal IJ ones. Flipping this order of parallelization increases the parallelism manyfold, which is necessary for high throughput on GPUs and other massively parallel architectures. FV3 parallelizes in this way because it is advantageous for their target architectures, which are CPUs with branch prediction and large caches. On CPU architectures instruction divergence inside the tight loops due to FV3's corner and edge handling is not an issue, but there is a limit to what can be done in a single-instruction, multiple-data (SIMD) GPU kernel. Because of their data dependencies these corner and edge operations cannot be executed in parallel with the rest of the code, requiring a separate kernel launch for each of them. This breaks what could be one large and optimized stencil into separate stencils separated by such corner calculations, though there are certain conditions under which we can fuse these operations into the same kernel.

This example shows how algorithmic choices have large impacts on performance. The choice of specialized corner and edge handling made for FV3 fits a CPU architecture well but is suboptimal for GPU architectures. Changes to the algorithm can be made to alleviate this problem, such as a duo-grid approach demonstrated in Chen (2021). Since Pace is intended to be a one-to-one port of FV3 in its current state, this is not a development we are focused on at the moment, though it could be a fruitful direction for other DSL models.

6 Pace in action

6.1 Driving the model

The driver code in Listing 3 showcases the power of Python as an API language for configuring, compiling, and running computational codes. The `command_line` entry function reads all information needed to run the model from disk and directly defines the behavior of the driver class. Within the driver class (not shown), the dynamical core and physics components only have access to the configuration and variables they use, which makes it much easier for a new developer of the model to understand what the code does. This is true all the way down through its components, as described in Sect. 3.1.

This example also showcases the advantage of working in a widely used language with a vibrant open-source package ecosystem. Python has powerful tools available for defining command-line interfaces. Taking a yaml configuration file and mapping it onto a nested configuration class is as simple as using pyyaml and dacite, which provide this functionality. Python's built-in datetime and timedelta types make it easy to manage model execution time, and external packages such as cftime provide support for a wide range of calendars. Diagnostics storage makes use of the xarray and zarr packages to greatly simplify the code we need to write. This well-established ecosystem of tools maintained by scientists and engineers from a range of disciplines and sectors is incredibly helpful when developing an atmospheric model.

And in turn, having a model written in Python means the tools we have written can be used by anyone who uses the language. This is particularly important given the popularity of Python as a language for scientific analysis. When processing the output of Pace, a scientist has direct access to the tools and numerical code used by the model itself.

6.2 Use cases

One of the advantages of Python is the ecosystem surrounding it. For example, packages such as NumPy and SciPy make it easy to perform common mathematical and numerical operations and manipulate data; Matplotlib enables interactive visualization and creates static or animated images; a Jupyter Notebook allows users to create and share their computational documents; TensorFlow provides end-to-end libraries for machine learning and artificial intelligence applications. With Pace, we can leverage all these tools to make running, processing, and visualizing climate model output all in one Python script.

Pace also enables novel workflows through the use of Jupyter Notebooks. The Pace repository contains an example notebook running one component of Pace. We first initialize a cubed-sphere grid and an idealized atmospheric state, and then we run the tracer advection operation and visualize the results. In this case the analytic zonal wind profile should advect the tracer mass once around the Earth in 12 model days, and any deviation from this indicates a problem with the advection code. This workflow is meant to mimic how a model developer could develop, implement, and debug a new advection scheme. We take advantage of the fact that each component in Pace is modular and skip over running the entire model, which enables rapid prototyping without leaving the Python ecosystem. Beyond development, this capability is useful for teaching, allowing students to inspect elements of the model individually.

Another benefit of Pace is that it facilitates incorporating novel applications and approaches into the model, such as machine learning emulation. Oftentimes machine learning in

```python
@click.command()
@click.argument(
    "CONFIG_PATH",
    required=True,
    type=click.Path(exists=True, readable=True, dir_okay=False, resolve_path=True),
)
@click.option(
    "--log-rank",
    type=click.INT,
    help="rank to log from, or all ranks by default, ignored if running without MPI",
)
@click.option(
    "--log-level",
    default="info",
    help="one of 'debug', 'info', 'warning', 'error', 'critical'",
)
def command_line(config_path: str, log_rank: Optional[int], log_level: str):
    """
    Run the driver.

    CONFIG_PATH is the path to a DriverConfig yaml file.
    """
    configure_logging(log_rank=log_rank, log_level=log_level)
    logger.info("loading DriverConfig from yaml")
    with open(config_path, "r") as f:
        config = yaml.safe_load(f)
        driver_config = DriverConfig.from_dict(config)
    logging.info(f"DriverConfig loaded: {yaml.dump(dataclasses.asdict(driver_config))}")
    main(driver_config=driver_config)
```

Listing 3. The driver function to run Pace from the command line. The `command_line` function is able to gather the configuration and set logging behavior and run the model.

```python
class Physics:
    ...

    # GT4Py stencil-based physics:
    prepare_microphysics(physics_state)
    microph_state = physics_state.microphysics
    microphysics(microph_state)
    ...

    # Machine-learning based microphysics
    emulation_model = tf.keras.models.load_model("microphysics.tf")

    emulation_dict = prepare_emulation_data(physics_state.microphysics)
    predictions = emulation_model(emulation_dict)
    model_outputs = unpack_predictions(predictions, emulation_model.output_names, _)
    update_physics(physics_state, model_outputs)
```

Listing 4. Pseudo-code outlining how a machine-learned microphysics emulation scheme could be incorporated as a model component in Pace.

climate models requires complicated workflows such as calling Fortran from Python (McGibbon et al., 2021) or calling Python from Fortran. Because Pace is written in Python, we can train and execute an machine learning (ML) model directly, bypassing difficult infrastructure to pass variables between languages. In Listing 4 we show Pace's ability to replace a stencil-based microphysics scheme with a pre-trained TensorFlow-based microphysics emulator. Taking advantage of the modularity and separation of initialization and call time of the individual module, we simply load the TensorFlow model during the initialization of the Physics class and replace the call signature of microphysics. Figure 10 shows an example of the emulator applied to the baroclinic instability test case.

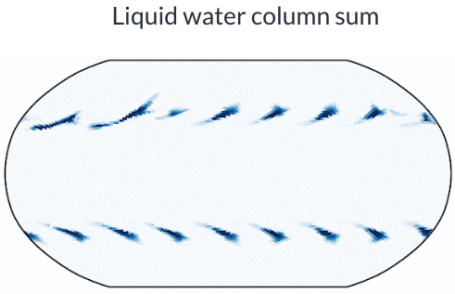

Figure 10. Microphysics emulation results: liquid water column sum of a baroclinic instability simulation on day 10 at 200 km resolution.

7 Limitations

Our research has focused on porting a subset of NOAA's FV3GFS model so as to limit the porting effort and focus on framework development. In particular, the nesting capabilities of FV3 were not ported. Likewise, we forked the model code early in the project and did not pull any updates added to the original Fortran code by NOAA's team. Despite this limiting choice, we believe the two namelist configurations we ported cover a wide range of applications to show that any model code can be effectively ported.

While this paper presents our port of the dynamical core and microphysics, we are actively working on integrating and validating GT4Py ports of the full GFS physics suite. Our results integrating the GFDL microphysics scheme (Chen and Lin, 2013) into Pace not only show validation and competitive performance, but also demonstrate the feasibility of implementing physics parameterizations using the same strategy deployed for the dynamical core. We have written GT4Py implementations of the remaining physics schemes (PBL, turbulence, shallow convection, sea ice, land surface, and radiation), and work is ongoing to optimize and integrate them into the Pace model.

Only the baroclinic wave test case was tested at the larger 150-rank configuration described in Sect. 4.2. A subset of an earlier version of the code was run on 2400 GPU nodes (Ben-Nun et al., 2022) before achieving full validation, alleviating concerns about distributed performance.

Lastly, as explained in Sect. 5, we have only focused on GPU performance optimization thus far, leaving CPU performance suboptimal. We are confident that CPU optimization is achievable within the boundary of the current framework and only requires careful engineering.

8 Conclusions

We have presented Pace, an open-source performance-portable implementation of the FV3 dynamical core and GFDL cloud microphysics written in Python using the GT4Py domain-specific language. The DSL implementation allows Pace to run on both CPU and GPU architectures, where it achieves high performance. The use of Python as a front-end language lets us write Pace in a modular, productive style and gives us access to Python's powerful testing tools, which allows us to debug and validate Pace.

We have demonstrated our method of porting code from Fortran to Python, building Pace hierarchically from small units of ported code that validate regularly against their Fortran equivalents. Fully porting Pace was a combination of porting the code to the DSL and extending the DSL to cover new algorithmic motifs required by the model, such as "while loops" inside stencils and allowing computations to occur on horizontal subdomains. Our testing strategy ensures that our code remains equivalent to the Fortran model throughout our development, including front-end and back-end changes. Our approach can be adopted to reimplementing other model codes and provides a good template for porting weather and climate models between languages.

We have shown the performance implications of the DSL design and implementation, leading to a $\sim 3.3\times$ speedup for Pace's GPU back end over the Fortran reference. One advantage of Python is the blending of compile time and runtime allows us to compile the model for a specific runtime configuration. Increasing the amount of code exposed to the DSL compiler had a strong impact on our model performance and drove us to adopt DaCe to leverage full-program optimizations. Algorithmic changes such as the duo-grid implementation can drive further optimization.

Pace takes great advantage of the Python ecosystem. Pace has full access to Python packages such as NumPy (C. R. Harris et al., 2020) and Matplotlib (Hunter, 2007). We can run the model, or just a model component, in a Jupyter Notebook, opening new avenues for model exploration and debugging. The Python front end also allows Pace to easily incorporate and couple to machine-learned model components, such as physics parameterizations. We have also illustrated the relative ease of debugging Pace, both through Python's developer tools and through the ability to implement novel tests like subtracting two dycores to determine whether temporary variables are properly reset between time steps.

As supercomputer heterogeneity increases, Pace stands as both a useful atmospheric model and a strong proof of concept for the DSL approach to performance portability. We have shown the advantages of an atmospheric model written in a high-level language such as Python, with further development still to come. We hope to see this approach adopted more broadly in the modeling community.

Code and data availability. Code releases for Pace (George et al., 2022, https://doi.org/10.5281/zenodo.7464843) and GT4Py (Dahm et al., 2022, https://doi.org/10.5281/zenodo.7080260) are available on Zenodo.

Author contributions. JD, ED, FD, OE, RG, JM, TW, and EW: equal contributions to methodology, software, validation, and writing. CK: software and validation. TBN and LG: methodology, software, and validation. LH: validation, writing, and supervision. OF: conceptualization, methodology, writing, project administration, supervision, and funding acquisition.

Competing interests. The contact author has declared that none of the authors has any competing interests.

Disclaimer. Publisher's note: Copernicus Publications remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Acknowledgements. We acknowledge the contributions of Mark Cheeseman, Yannick Niedermayr, and Mikael Stellio to the Pace code base, domain partitioning code, and the port of the microphysics, respectively. We thank Rusty Benson (NOAA/GFDL) for insightful discussions. Further, we acknowledge contributions from the whole GT4Py team, specifically Hannes Vogt (CSCS) and Enrique Gonzalez (CSCS), for their help in implementing a validating version of FV3 using GT4Py. We also thank two anonymous reviewers for their constructive suggestions.

Financial support. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID s1053. We thank Vulcan Inc., the Allen Institute for Artificial Intelligence (AI2), the Geophysical Fluid Dynamics Laboratory (GFDL) of NOAA, and the Global Modeling and Assimilation Office (GMAO) of NASA for supporting this work. Tal Ben-Nun is supported by the Swiss National Science Foundation (Ambizione project no. 185778).

Review statement. This paper was edited by Travis O'Brien and reviewed by two anonymous referees.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, Zenodo [software], https://doi.org/10.5281/zenodo.4724125, 2015.

Adams, S., Ford, R., Hambley, M., Hobson, J., Kavčič, I., Maynard, C., Melvin, T., Müller, E., Mullerworth, S., Porter, A., Rezny, M., Shipway, B., and Wong, R.: LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models, J. Parallel Distr. Com., 132, 383–396, https://doi.org/10.1016/j.jpdc.2019.02.007, 2019.

Alnaes, M. S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., and Rognes, M. E., and Wells, G. N.: The FEniCS Project Version 1.5, Archive of Numerical Software [software], https://doi.org/10.11588/ans.2015.100.20553, 2015.

Augier, P., Bolz-Tereick, C. F., Guelton, S., and Mohanan, A. V.: Reducing the ecological impact of computing through education and Python compilers, Nature Astronomy, 5, 334–335, https://doi.org/10.1038/s41550-021-01342-y, 2021.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K.: Cython: The Best of Both Worlds, Comput. Sci. Eng., 13, 31–39, https://doi.org/10.1109/MCSE.2010.118, 2011.

Ben-Nun, T., de Fine Licht, J., Ziogas, A. N., Schneider, T., and Hoefler, T.: Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, November 2019, 81, 1–14, https://doi.org/10.1145/3295500.3356173, 2019.

Ben-Nun, T., Groner, L., Deconinck, F., Wicky, T., Davis, E., Dahm, J., Elbert, O. D., George, R., McGibbon, J., Trümper, L., Wu, E., Fuhrer, O., Schulthess, T., and Hoefler, T.: Productive Performance Engineering for Weather and Climate Modeling with Python, arXiv, https://doi.org/10.48550/ARXIV.2205.04148, 2022.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q.: JAX: composable transformations of Python+NumPy programs, http://github.com/google/jax (last access: September 2022), 2018.

Brenowitz, N. D. and Bretherton, C. S.: Spatially extended tests of a neural network parametrization trained by coarse-graining, J. Adv. Model. Earth Sy., 11, 2728–2744, 2019.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D.: Language Models are Few-Shot Learners, in: Advances in Neural Information Processing Systems, edited by: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., vol. 33, pp. 1877–1901, Curran Associates, Inc., https://doi.org/10.48550/arXiv.2005.14165, 2020.

Burns, K. J., Vasil, G. M., Oishi, J. S., Lecoanet, D., and Brown, B. P.: Dedalus: A flexible framework for numerical simulations with spectral methods, Phys. Rev. Research, 2, 023068, https://doi.org/10.1103/PhysRevResearch.2.023068, 2020.

Chen, J.-H. and Lin, S.-J.: Seasonal predictions of tropical cyclones using a 25-km-resolution general circulation model, J. Climate, 26, 380–398, 2013.

Chen, X.: The LMARS Based Shallow-Water Dynamical Core on Generic Gnomonic Cubed-Sphere Geometry, J. Adv. Model. Earth Sy., 13, e2020MS002280, https://doi.org/10.1029/2020MS002280, 2021.

Clement, V., Marti, P., Lapillonne, X., Fuhrer, O., and Sawyer, W.: Automatic Port to OpenACC/OpenMP for Physical Parameterization in Climate and Weather Code Using the CLAW Compiler, Supercomputing Frontiers and Innovations, 6, 51–63, https://doi.org/10.14529/jsfi190303, 2019.

Dahm, J., Groner, L., Paredes, E. G., Thaler, F., Vogt, H., Davis, E., Haeuselmann, R., Ehrengruber, T., Ubbiali, S., Wicky, T., Deconinck, F., Ben-Nun, T., and George, R.: ai2cm/gt4py: v0.1.0 GMD release, Zenodo [code], https://doi.org/10.5281/zenodo.7080260, 2022.

E3SM Project, DOE: Energy Exascale Earth System Model v2.0, Computer Software, GitHub [software], https://doi.org/10.11578/E3SM/dc.20210927.1, 2021.

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 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0, Geosci. Model Dev., 11, 1665–1681, https://doi.org/10.5194/gmd-11-1665-2018, 2018.

George, R., Wu, E., McGibbon, J., Dahm, J., Davis, E., Wicky, T., Deconinck, F., Kung, C., Fuhrer, O., Elbert, O., Savarin, A., Brenowitz, N. D., Cheeseman, M., Henn, B., Clark, S., and Niedermayr, Y.: ai2cm/pace: v0.2.0, Zenodo [code and data set], https://doi.org/10.5281/zenodo.7464843, 2022.

Giorgetta, M. A., Sawyer, W., Lapillonne, X., Adamidis, P., Alexeev, D., Clément, V., Dietlicher, R., Engels, J. F., Esch, M., Franke, H., Frauen, C., Hannah, W. M., Hillman, B. R., Kornblueh, L., Marti, P., Norman, M. R., Pincus, R., Rast, S., Reinert, D., Schnur, R., Schulzweida, U., and Stevens, B.: The ICON-A model for direct QBO simulations on GPUs (version icon-cscs:baf28a514), Geosci. Model Dev., 15, 6985–7016, https://doi.org/10.5194/gmd-15-6985-2022, 2022.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E.: Array programming with NumPy, Nature, 585, 357–362, https://doi.org/10.1038/s41586-020-2649-2, 2020.

Harris, L., Zhou, L., Lin, S.-J., Chen, J.-H., Chen, X., Gao, K., Morin, M., Rees, S., Sun, Y., Tong, M., Xiang, B., Bender, M., Benson, R., Cheng, K.-Y., Clark, S., Elbert, O. D., Hazelton, A., Huff, J. J., Kaltenbaugh, A., Liang, Z., Marchok, T., Shin, H. H., and Stern, W.: GFDL SHiELD: A Unified System for Weather-to-Seasonal Prediction, J. Adv. Model. Earth Sy., 12, e2020MS002223, https://doi.org/10.1029/2020MS002223, 2020.

Harris, L., Chen, X., Putman, W., Zhou, L., and Chen, J.-H.: A Scientific Description of the GFDL Finite-Volume Cubed-Sphere Dynamical Core, technical Memorandum, https://repository.library.noaa.gov/view/noaa/30725 (last access: September 2022), 2021.

Harris, L. M. and Lin, S.-J.: A Two-Way Nested Global-Regional Dynamical Core on the Cubed-Sphere Grid, Mon. Weather Rev., 141, 283–306, https://doi.org/10.1175/MWR-D-11-00201.1, 2013.

Hunter, J. D.: Matplotlib: A 2D graphics environment, Comput. Sci. Eng., 9, 90–95, https://doi.org/10.1109/MCSE.2007.55, 2007.

Jablonowski, C. and Williamson, D. L.: A baroclinic instability test case for atmospheric model dynamical cores, Q. J. Roy. Meteor. Soc., 132, 2943–2975, https://doi.org/10.1256/qj.06.12, 2006.

Lam, S. K., Pitrou, A., and Seibert, S.: Numba: A LLVM-Based Python JIT Compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, Association for Computing Machinery, New York, NY, USA, November 2015, 7, 1–6, https://doi.org/10.1145/2833157.2833162, 2015.

Lapillonne, X., Osterried, K., and Fuhrer, O.: Chapter 13 – Using OpenACC to port large legacy climate and weather modeling code to GPUs, in: Parallel Programming with OpenACC, edited by: Farber, R., pp. 267–290, Morgan Kaufmann, Boston, https://doi.org/10.1016/B978-0-12-410397-9.00013-5, 2017.

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?, Geosci. Model Dev., 11, 1799–1821, https://doi.org/10.5194/gmd-11-1799-2018, 2018.

Lin, S.-J.: A "Vertically Lagrangian" Finite-Volume Dynamical Core for Global Models, Mon. Weather Rev., 132, 2293–2307, https://doi.org/10.1175/1520-0493(2004)132<2293:AVLFDC>2.0.CO;2, 2004.

Lin, S.-J. and Rood, R. B.: Multidimensional Flux-Form Semi-Lagrangian Transport Schemes, Mon. Weather Rev., 124, 2046–2070, https://doi.org/10.1175/1520-0493(1996)124<2046:mffslt>2.0.co;2, 1996.

McGibbon, J., Brenowitz, N. D., Cheeseman, M., Clark, S. K., Dahm, J. P. S., Davis, E. C., Elbert, O. D., George, R. C., Harris, L. M., Henn, B., Kwa, A., Perkins, W. A., Watt-Meyer, O., Wicky, T. F., Bretherton, C. S., and Fuhrer, O.: fv3gfs-wrapper: a Python wrapper of the FV3GFS atmospheric model, Geosci. Model Dev., 14, 4401–4409, https://doi.org/10.5194/gmd-14-4401-2021, 2021.

Méndez, M., Tinetti, F. G., and Overbey, J. L.: Climate Models: Challenges for Fortran Development Tools, in: 2014 Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, New Orleans, LA, USA, 21 November 2014, 6–12, https://doi.org/10.1109/SE-HPCCSE.2014.7, 2014.

Monteiro, J. M., McGibbon, J., and Caballero, R.: sympl (v. 0.4.0) and climt (v. 0.15.3) – towards a flexible framework for building model hierarchies in Python, Geosci. Model Dev., 11, 3781–3794, https://doi.org/10.5194/gmd-11-3781-2018, 2018.

Partee, S., Ellis, M., Rigazzi, A., Bachman, S., Marques, G., and Shao, A.: SmartSim: Online Analytics and Machine Learning for HPC Simulations, Zenodo [code], https://doi.org/10.5281/zenodo.4986182, 2021a.

Partee, S., Ellis, M., Rigazzi, A., Bachman, S., Marques, G., Shao, A., and Robbins, B.: Using Machine Learning at Scale in HPC Simulations with SmartSim: An Application to Ocean Climate Modeling, arXiv, https://doi.org/10.48550/ARXIV.2104.09355, 2021b.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library, in: Advances in Neural Information Processing Systems 32, edited by: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., Curran Associates, Inc., https://doi.org/10.48550/arXiv.1912.01703, pp. 8024–8035, 2019.

Putman, W. M. and Lin, S.-J.: Finite-volume transport on various cubed-sphere grids, J. Comput. Phys., 227, 55–78, https://doi.org/10.1016/j.jcp.2007.07.022, 2007.

Strohmaier, E., Meuer, H. W., Dongarra, J., and Simon, H. D.: The TOP500 List and Progress in High-Performance Computing, Computer, 48, 42–49, https://doi.org/10.1109/MC.2015.338, 2015.

Theis, T. N. and Wong, H.-S. P.: The End of Moore's Law: A New Beginning for Information Technology, Comput. Sci. Eng., 19, 41–50, https://doi.org/10.1109/MCSE.2017.29, 2017.

Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J.: Kokkos 3: Programming Model Extensions for the Exascale Era, IEEE T. Parall. Distr., 33, 805–817, https://doi.org/10.1109/TPDS.2021.3097283, 2022.

van den Oord, G., Jansson, F., Pelupessy, I., Chertova, M., Grönqvist, J. H., Siebesma, P., and Crommelin, D.: A Python interface to the Dutch atmospheric large-eddy simulation, SoftwareX, 12, 100608, https://doi.org/10.1016/j.softx.2020.100608, 2020.

Vila-Pérez, J., Van Heyningen, R. L., Nguyen, N.-C., and Peraire, J.: Exasim: Generating discontinuous Galerkin codes for numerical solutions of partial differential equations on graphics processors, SoftwareX, 20, 101212, https://doi.org/10.1016/j.softx.2022.101212, 2022.

Zhou, L., Lin, S.-J., Chen, J.-H., Harris, L. M., Chen, X., and Rees, S. L.: Toward Convective-Scale Prediction within the Next Generation Global Prediction System, B. Am. Meteorol. Soc., 100, 1225–1243, https://doi.org/10.1175/BAMS-D-17-0246.1, 2019.