Productive Performance Engineering for Weather and Climate Modeling with Python

Tal Ben-Nun[*], Linus Groner[†], Florian Deconinck[‡], Tobias Wicky[‡], Eddie Davis[‡], Johann Dahm[‡],
Oliver D. Elbert[‡], Rhea George[‡], Jeremy McGibbon[‡], Lukas Trümper[*], Elynn Wu[‡],
Oliver Fuhrer[‡], Thomas Schulthess[†] and Torsten Hoefler[*]
[*]Department of Computer Science
ETH Zürich, 8092 Zürich, Switzerland
Email: {talbn, lukashans.truemper, htor}@inf.ethz.ch
[†]Swiss National Supercomputing Centre (CSCS), 6900 Lugano, Switzerland
Email: {linus.groner, schulthess}@cscs.ch
[‡]Allen Institute for Artificial Intelligence, Seattle, Washington 98103, USA
Email: {floriand, tobiasw, eddied, johannd, olivere, rheag, jeremym, elynnw, oliverf}@allenai.org

***Abstract*—Earth system models are developed with a tight coupling to target hardware, often containing specialized code predicated on processor characteristics. This coupling stems from using imperative languages that hard-code computation schedules and layout. We present a detailed account of optimizing the Finite Volume Cubed-Sphere Dynamical Core (FV3), improving productivity and performance. By using a declarative Python-embedded stencil domain-specific language and data-centric optimization, we abstract hardware-specific details and define a semi-automated workflow for analyzing and optimizing weather and climate applications. The workflow utilizes both local and full-program optimization, as well as user-guided fine-tuning. To prune the infeasible global optimization space, we automatically utilize repeating code motifs via a novel transfer tuning approach. On the Piz Daint supercomputer, we scale to 2,400 GPUs, achieving speedups of up to 3.92× over the tuned production implementation at a fraction of the original code.**

***Index Terms*—Numerical Weather Prediction, Python, Data-Centric Programming**

I. Introduction

Climate change and the associated weather extremes is one of the biggest challenges facing humanity today. The basis of what we know about our future stem from simulations using weather and climate models running on some of the world's largest supercomputing infrastructures. But progress in leveraging the power of current and emerging supercomputing hardware architectures is slow, due to legacy software engineering and lift-and-shift approaches to code portability.

In order to address the shortcomings of currently available weather and climate data, there is an urgent need to address this software productivity gap to enable simulations with higher fidelity [1]. Given today's hardware landscape and increasing complexity of models, the current approach is not sustainable [2]. We urgently need to enable more rapid development cycles and faster adoption of leadership class supercomputing infrastructures [3].

In this paper, we present a novel approach to productive performance engineering for weather and climate modeling using Python. Our approach, summarized in Fig. 1, increases developer productivity while not making any compromises in terms of performance and performance portability.

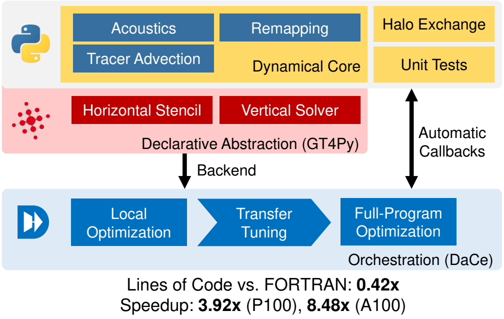


Fig. 1: System overview.

To enable performance engineering, we must express the code in a way that allows it to mutate schedules (i.e., work distribution among processors and order of operations) while maintaining the algorithms. To this end, we leverage a domain-specific language (DSL) embedded in Python, called GridTools for Python (GT4Py), which allows the developer to express the algorithms on a high level of abstraction. The Python programming language and package ecosystem allows for writing modular, re-usable code which can easily be unit-tested (Section V). The pure-Python backend of the DSL toolchain is ideal for rapid prototyping, debugging and interactive visualization of algorithmic approaches. The abstraction and domain-specificity of the DSL allows for concise, declarative code which is not littered with hardware dependent optimizations or annotations.

For the actual performance engineering we leverage a data-centric (DaCe) parallel programming framework as a backend for optimization and code generation (Section IV-B and VI). This framework allows to modify schedules, data layouts, and memory placement (Section VII). We take a disciplined model-driven approach for optimization [4], which keeps track of performance bounds (e.g., memory bandwidth) during local and global optimizations to guide further decisions.

Furthermore, we introduce a novel automatic tuning tech-

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

nique called transfer tuning (Section VII-B). The core problem of automatic tuning at scale is the high rate of configurations that must be traversed. Since many optimizations recur throughout the program, good configurations are already known from earlier decisions. Transfer tuning therefore extracts the top configurations for each type of transformation from a smaller part of the application and applies matching patterns to the others, significantly pruning the search space.

To illustrate our approach we choose the Finite-Volume Cubed-Sphere Dynamical Core (FV3), an open-source solver implemented in FORTRAN. FV3 is used as the workhorse of several popular community weather and climate models. It was specifically designed to reap the benefits of multicore CPUs [5], [6] by keeping two-dimensional fields for most of the time step, thereby exhibiting high cache utilization.

We port FV3 to GPUs, whose microarchitecture and programming model do not share the same cache benefits. With the DSL and global optimization, we gain a 3.9× speedup at scale on the Piz Daint supercomputer. As the methodology applies to any model, it serves as a testament that Python-based models can be feasible, productive, and fast.

The paper makes the following contributions:

- Methodology for guided weather and climate performance engineering, defining the domain-specific search space for local and global optimization.
- Performance modeling tools for said methodology to bound performance (i.e., indicate when to stop optimizing certain components) and guide tuning when using data-centric Python and declarative stencil definitions.
- *Transfer Tuning*, a novel technique that drastically reduces auto-tuning cost in large applications by reusing beneficial patterns.
- Demonstrating the methodology on the FV3 weather model, where for the first time a high-performance, **full dynamical core** is entirely written in, compiled from, and run on up to 2,400 GPUs within Python. This includes a comprehensive account of the involved optimizations, performance upper bounds for the sizes used in operational forecasting, and effort estimation for porting the approach to new hardware or weather models.

II. The Finite-Volume Cubed-Sphere (FV3) model

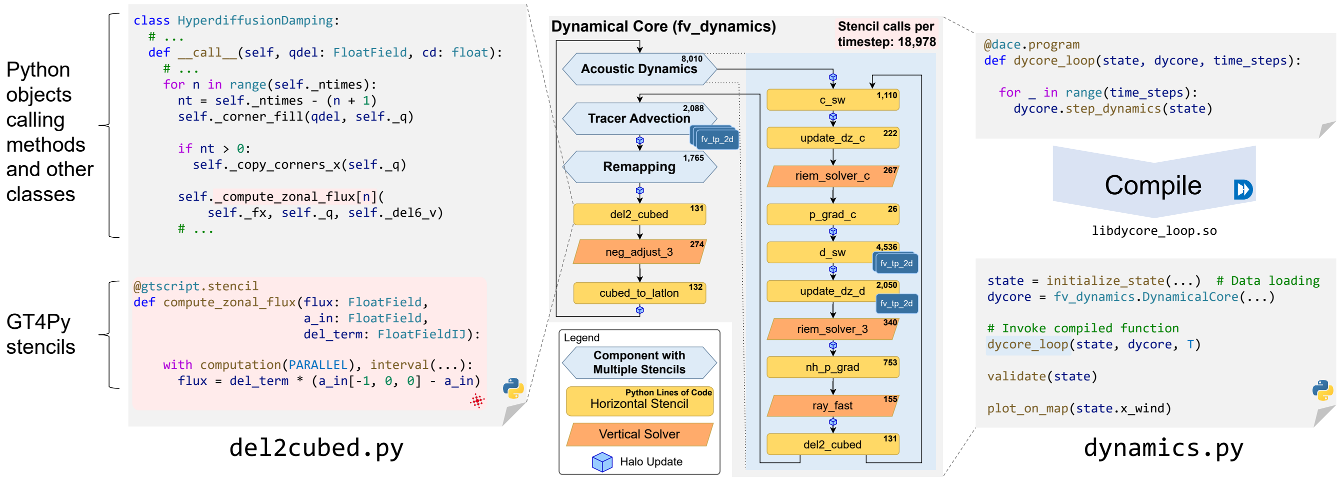
The *dynamical core* is the foundation of any weather and climate model. It integrates the governing equations of air flow and thermodynamics forward in time from a given initial state. Finite-Volume Cubed-Sphere Dynamical Core (FV3) is a dynamical core developed at the Geophysical Fluid Dynamics Laboratory (GFDL) and has been chosen as the dynamical core for the Next Generation Global Prediction System project [7]. FV3 has been deployed in numerous community models (e.g., SHiELD, GEOS, UFS, CESM) both in production and research and has shown promising performance in speed and accuracy on the global and regional scale [8].

While FV3 is capable of solving both the hydrostatic and non-hydrostatic set of governing equations, the focus of this paper will be on the non-hydrostatic option. In particular, FV3 solves the fully compressible Euler equations on the gnomonic cubed-sphere grid and a Lagrangian vertical coordinate [5], [6], [9]. It is fully explicit in the time integration except for fast vertically propagating sound and gravity waves. FV3 is designed to be computationally efficient and many algorithms have been selected or revised for optimal efficiency. In fact, efficient implementation of the scientific algorithms is one of the defining traits of FV3. The reference of FV3 is implemented in FORTRAN and parallelized using OpenMP directives for on-node threading and a two-dimensional domain decomposition in the horizontal dimensions using MPI library calls.

FV3 utilizes sub-stepping to integrate different processes on different timesteps. There are three levels of sub-stepping: the **physics timestep** is the outermost loop and advances the model by an atmospheric timestep in every iteration. Each iteration contains calls to the dynamical core as well as potential calls to desired physical parametrizations. As this work is focused on the dynamical core, we leave any physics packages out of the analysis. The **remapping timestep** is the middle loop found in the model (see Fig. 2, middle, for an overview of the structure of the remapping timestep). The number of substeps can be tuned to balance performance and model stability. In each remapping timestep the model advects tracer variables (middle hexagon in Fig. 2), remaps the deformed Lagrangian to the reference “Eulerian” vertical coordinate levels (bottom hexagon) and contains a call to the innermost loop, where the integration of the dynamical equations happens on Lagrangian surfaces. The number of iterations of the innermost loop of the model, the **acoustic substep** (top hexagon), is determined by horizontal sound wave propagation and thus directly related to the grid spacing. In this innermost loop, there are several execution points where nonblocking halo exchanges of one or more fields occur.

The FORTRAN implementation of the model has been heavily optimized for multi-core CPU architectures. The structure of the solver lends itself to leveraging data-locality in the cache hierarchy of the CPU, since large blocks of code act solely on horizontal planes of the data fields. In the strong scaling regime, sub-domains assigned to a single MPI process are small enough so multiple two-dimensional horizontal planes fit into an L2 cache. In these parts of the code (many of the yellow boxes in Fig. 2), the vertical loop has been hoisted outwards as far as possible (in a strategy called K-blocking). This strategy is a good choice for this specific dynamical core as there is very limited vertical coupling throughout the model.

As part of the first intercomparison project of global storm-resolving models (GSRMs) – the DYnamics of the Atmospheric general circulation Modeled On Non-hydrostatic Domains (DYAMOND) [10] – FV3 demonstrated the capability of running a global simulation at 3.3 km horizontal resolution at 19 simulation days per day using 384 nodes (dual-socket, Intel Broadwell, 18 cores/socket) and produced remarkable similarities for a snapshot comparison against satellite observations from Himawari 8, making itself a top contender of GSRMs in the world.



```python
class HyperdiffusionDamping:
  # ...
  def __call__(self, qdel: FloatField, cd: float):
    # ...
    for n in range(self._ntimes):
      nt = self._ntimes - (n + 1)
      self._corner_fill(qdel, self._q)

      if nt > 0:
        self._copy_corners_x(self._q)

      self._compute_zonal_flux[n](
          self._fx, self._q, self._del6_v)
      # ...
```

```python
@gtscript.stencil
def compute_zonal_flux(flux: FloatField,
                       a_in: FloatField,
                       del_term: FloatFieldIJ):

    with computation(PARALLEL), interval(...):
      flux = del_term * (a_in[-1, 0, 0] - a_in)
```

```python
@dace.program
def dycore_loop(state, dycore, time_steps):

  for _ in range(time_steps):
    dycore.step_dynamics(state)
```

```python
state = initialize_state(...)  # Data loading
dycore = fv_dynamics.DynamicalCore(...)

# Invoke compiled function
dycore_loop(state, dycore, T)

validate(state)

plot_on_map(state.x_wind)
```

Fig. 2: Code structure of the FV3 dynamical core. The entire codebase is written in object-oriented Python, where stencils use the GridTools for Python (GT4Py) embedded DSL. The time-stepping loop is then compiled to a library using the DaCe framework, and invoked as a single call to avoid Python interpreter overhead.

III. FV3 Optimization Efforts

Recently, the push towards a km-scale climate model is advocated by the community to commensurate with the challenges posed by climate change [11]. High resolution simulation has the ability to better represent weather phenomena such as deep convection, which remains a source of uncertainty in climate predictions. However, it is difficult to run such simulations at a high enough throughput without major change in programming paradigms (i.e., the ability to run climate models efficiently on current and emerging computer hardware with hybrid nodes) [1]. To this end, several directions were researched in making FV3 performance portable.

A. *Domain-Specific Languages for Weather and Climate*

While parts of FV3 were optimized for other hardware targets, the full model was not ported to a different architecture. Several efforts [12], [13] have shown that parts of FV3 can run faster on GPUs than CPUs using CUDA FORTRAN [14] or compiler directive (e.g., OpenACC) approaches. This, however, creates hardware-specific code that is unlikely to run with good performance on other architectures, and is difficult to read and maintain when compiler libraries and requirements change. Additionally, the prevalent algorithmic motifs in atmospheric models are a limited subset of numerical operations, mostly consisting of stencil computations on 3D arrays and columns, and thus are well suited to specializing optimizations that can outperform general-purpose compilers.

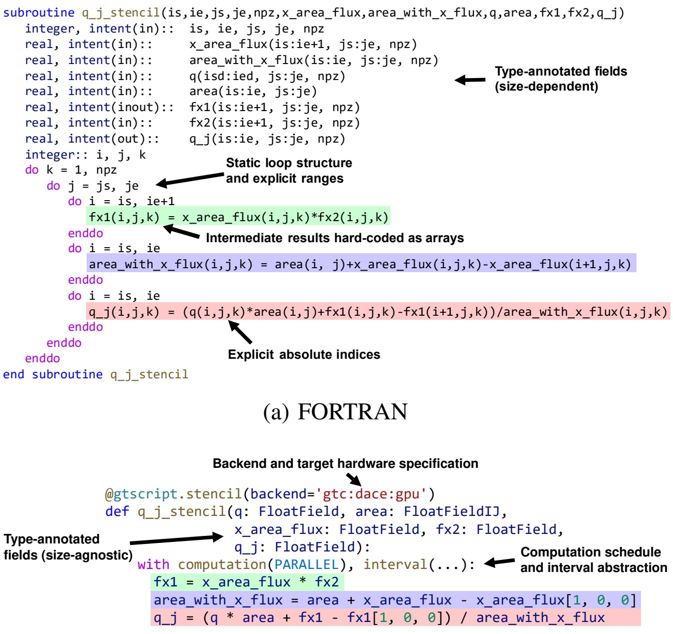
Using a DSL expands the possibilities for both how to express and how to optimize FV3 code. The three P's (productivity, portability, performance) are aspirational concepts for application sustainability [15] that DSLs can readily address. By creating a layer of abstraction to represent the algorithms of the model separate from the code optimized for a specific hardware, the abstracted code can be expressed in a way that is similar to the discretized mathematical equations, so that working on the algorithms is straightforward.

Several DSLs targeting atmospheric applications demonstrate the validity of this approach. CLAW [16] is a FORTRAN-based DSL supporting patterns commonly found in weather and climate applications and has been shown to be effective for certain parts of weather and climate models. DAWN [17] and the Open Earth Compiler [18] are compiler toolchains for weather and climate applications that provide abstractions between a high level expression of algorithm while generating efficient code for multiple platforms. Julia has also been used to construct models [19]. As for feasibility of using a DSL for an entire model, the UK Met Office recently demonstrated deploying an entire dynamical core over the LFRic toolchain [20].

B. *Why Python as the driver language*

Python is a popular choice of language in Earth system sciences, particularly for processing, analyzing and visualizing Earth system model output. Python has many well-documented libraries available for this type of analysis, ranging from machine learning libraries such as Scikit-learn [21] and TensorFlow [22] to domain-specific libraries such as xarray [23] or Cartopy [24]. Additionally, its popularity in introductory programming courses make it have a very low barrier of entry for most new researchers in their standard workflow.

However, Earth system models are generally not written in Python. Data are typically saved to the filesystem by a compiled language such as FORTRAN before being post-processed in a Python script. Writing data to and reading it from the filesystem can be limiting due to storage constraints, and increases processing time. Furthermore, modifying the behavior of the model often means writing code in the compiled language, restricting the ability to use Python libraries. This is particularly an issue when trying to improve Earth system models with online machine learning, given the popularity of Python for training learned models [25].



```fortran
subroutine q_j_stencil(is,ie,js,je,npz,x_area_flux,area_with_x_flux,q,area,fx1,fx2,q_j)
   integer, intent(in)::  is, ie, js, je, npz
   real, intent(in)::     x_area_flux(is:ie+1, js:je, npz)
   real, intent(in)::     area_with_x_flux(is:ie, js:je, npz)
   real, intent(in)::     q(isd:ied, js:je, npz)
   real, intent(in)::     area(is:ie, js:je)
   real, intent(inout)::  fx1(is:ie+1, js:je, npz)
   real, intent(in)::     fx2(is:ie+1, js:je, npz)
   real, intent(out)::    q_j(is:ie, js:je, npz)
   integer:: i, j, k
   do k = 1, npz
      do j = js, je
         do i = is, ie+1
            fx1(i,j,k) = x_area_flux(i,j,k)*fx2(i,j,k)
         enddo
         do i = is, ie
            area_with_x_flux(i,j,k) = area(i, j)+x_area_flux(i,j,k)-x_area_flux(i+1,j,k)
         enddo
         do i = is, ie
            q_j(i,j,k) = (q(i,j,k)*area(i,j)+fx1(i,j,k)-fx1(i+1,j,k))/area_with_x_flux(i,j,k)
         enddo
      enddo
   enddo
end subroutine q_j_stencil
```

(a) FORTRAN

```python
@gtscript.stencil(backend='gtc:dace:gpu')
def q_j_stencil(q: FloatField, area: FloatFieldIJ,
                x_area_flux: FloatField, fx2: FloatField,
                q_j: FloatField):
    with computation(PARALLEL), interval(...):
        fx1 = x_area_flux * fx2
        area_with_x_flux = area + x_area_flux - x_area_flux[1, 0, 0]
        q_j = (q * area + fx1 - fx1[1, 0, 0]) / area_with_x_flux
```

(b) GT4Py

Fig. 3: Stencil written in FORTRAN and its corresponding parallel stencil in GridTools for Python.

Earth system scientists have previously used multiple approaches to circumvent interoperability issues, including FORTRAN libraries to pass data between FORTRAN and Python [26], [27], wrapping a FORTRAN model as a Python library [25], [28], writing FORTRAN libraries that can run Python-trained machine learning models [29], [30], or writing the model itself in Python or Cython [31]–[33].

C. Python-driven FV3

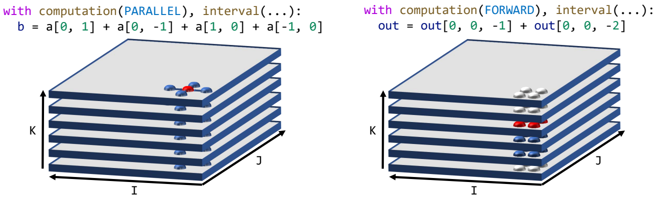
To circumvent issues with Python/FORTRAN interoperability, we chose to rewrite the FV3 dynamical core and its driver code entirely in Python, as summarized in Fig. 2. Each of the FV3 modules is situated in a class, which invokes stencils via a stencil DSL, as well as other modules that may contain stencils of their own (Fig. 2, left). The driver code is then automatically generated as C++/CUDA code and compiled to a shared library. The library handles all aspects, from memory management to distributed communication, and seamlessly integrates with the Python interpreter without changing the function signature (Fig. 2, right). This approach allows us to combine the flexibility and productivity of a scripted language with the performance gains from a compiled language.

IV. Workflow Components

To support our pure Python-driven approach, we use and extend the two Python-based frameworks described below.

A. GridTools for Python (GT4Py)

GT4Py [34] is an embedded DSL, which enables fast authoring of stencil codes for Earth system models. In GT4Py, stencils are written in a declarative manner, abstracting away most of the hardware-specific constructs while keeping the code close to its original mathematical formulation.



```python
with computation(PARALLEL), interval(...):
  b = a[0, 1] + a[0, -1] + a[1, 0] + a[-1, 0]
```

```python
with computation(FORWARD), interval(...):
  out = out[0, 0, -1] + out[0, 0, -2]
```

(a) Horizontal Stencil

(b) Vertical Solver (forward)

Fig. 4: Horizontal and vertical stencil abstraction.

GT4Py stencils are written as Python functions with a `@gtscript.stencil` decorator (Fig. 3). In a decorated function, statements can be grouped with a `computation` block that defines the stencil type, while `interval` statement restricts the computations to the specified vertical intervals. Within a single block, each assignment corresponds to a single stencil operation. Authoring a GT4Py stencil is thus a relatively direct translation of the original code. More detailed semantics can be found in GridTools [35].

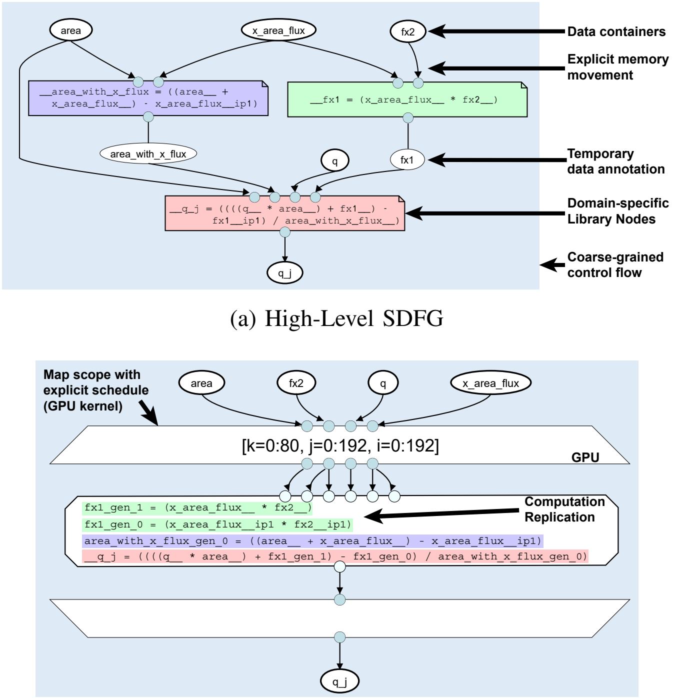
GT4Py classifies stencils into two types, as depicted in Fig. 4. Horizontal (parallel) Stencils contain no loop-carried dependencies — namely, the result of applying the stencil on one point cannot affect others — and Vertical Solvers, in which the output field is computed in a vertical order and can use previously computed values in higher (*forward* solvers) or lower (*backward*) vertical levels.

GT4Py does not define a specific domain size for its stencils, only dimensionality (e.g., `FloatFieldIJ` for 2D horizontal fields), and all accesses to memory in stencil operations use relative indices. Buffer sizes for fields are thus transparently defined by inferring halo regions and extents from usage in stencils. Loop order, data layout, and even fusion of multiple operations into one stencil are similarly abstracted away. The specified *backend* is responsible for optimization, memory allocation and layout, and code generation, enabling stencil-level decisions on scheduling.

B. Data-Centric Parallel Programming (DaCe)

The DaCe framework is an optimization infrastructure written in Python, which enables interactive analysis and optimization of data movement in programs. At the core of DaCe lies an intermediate representation (IR) called Stateful Dataflow Multigraphs (SDFG) [36].

The SDFG IR (Figures 5a and 5b) is composed of acyclic dataflow graphs nested in state machines, representing control flow. In each state, data containers (oval nodes) and data movement (edges) are explicitly defined separately from computations, which are represented by octagonal nodes. *Map* scopes (trapezoidal sections) represent parametric parallelism, replicating the graph within them for the indicated domain. Coarse-grained domain-specific computations (e.g., matrix multiplication) can be represented by *library nodes*, which can be expanded to “native” subgraphs containing the


area
x_area_flux
fx2
Data containers
Explicit memory movement
__area_with_x_flux = ((area__ + x_area_flux__) - x_area_flux__ip1)
__fx1 = (x_area_flux__ * fx2__)
area_with_x_flux
q
fx1
Temporary data annotation
__q_j = ((((q__ * area__) + fx1__) - fx1__ip1) / area_with_x_flux__)
Domain-specific Library Nodes
q_j
Coarse-grained control flow
Map scope with explicit schedule (GPU kernel)
area
fx2
q
x_area_flux
[k=0:80, j=0:192, i=0:192]
GPU
fx1_gen_1 = (x_area_flux__ * fx2__)
fx1_gen_0 = (x_area_flux__ip1 * fx2__ip1)
area_with_x_flux_gen_0 = ((area__ + x_area_flux__) - x_area_flux__ip1)
__q_j = ((((q__ * area__) + fx1_gen_1) - fx1_gen_0) / area_with_x_flux_gen_0)
Computation Replication
q_j

(a) High-Level SDFG

(b) Expanded and Fused SDFG

Fig. 5: Stateful Dataflow Multigraph (SDFG) that corresponds to the stencil from Fig. 3.

aforementioned components. This approach is widely used in DSLs that are lowered to SDFGs.

SDFGs inherently allow users to query data movement for exact ranges at any point of the program. This, in turn, enables *data-centric optimizations* in the form of graph rewriting rules. Examples include removing redundant memory allocation, creating local storage, scheduling map scopes to run on accelerators, change code schedule arbitrarily (e.g., tiling, fusion), and others. Information on removable (*transient*) containers is indicated on the graph.

DaCe can optimize programs written in different languages, including Python/NumPy [37], C [38], and embedded DSLs such as PyTorch [39]. The resulting SDFGs can then be optimized programmatically or interactively, and map to state-of-the-art code for a wide variety of hardware architectures, including multi-core CPUs, NVIDIA and AMD GPUs, and Xilinx and Intel FPGAs.

V. Python FV3 Implementation

The dynamical core of FV3 suits the two stencil classes that GT4Py targets well, but not exactly. Thus, some design decisions and concessions had to be made on both the model and the DSL to allow for this approach to work.

A. Object-oriented design

To maintain productivity, the following features should be possible to use without any performance penalty: (1) Python classes with complex inheritance; (2) function calls at arbitrary depths; (3) Pythonic objects such as lists and dictionaries; and (4) use of arbitrary amounts of temporary variables without worrying about memory allocation. We chose to value both maintainability as well as productivity via object-oriented programming (OOP) that represents each module of the dynamical core as its own class. Classes have fields, methods with clear inputs and outputs, and the required subset of the global configuration of the model.

TABLE I: Lines of Code (LoC) Comparison of FV3

Module Name	Python LoC	FORTRAN LoC
Dynamical Core	12,450	29,458
Finite Volume Transport	686	858
Riemann Solver C	253	267

As for the class structure, we keep the structure of the original FORTRAN subroutines (see Fig. 2) as modules in the Python implementation. As the FORTRAN model was designed with an emphasis on separating different numerical steps of the model, there was no need to change that.

One significant new opportunity this approach enables is modularized, fine-grained testing. We started the porting effort by saving input and output data for each of the target modules from the FORTRAN model. Our test suite has independent standalone unit-tests for model validation by comparing with the stored reference up to a given numerical precision. For stencils that do not have dependencies on other tiles of the cubed sphere, testing can even be performed on a single node in a “sequential” mode. All stencils have the option to be tested with any configuration of multiple subdomains on all six tiles supported by our standard partitioner. This fine-grained testing allows model developers to get rapid feedback on model behavior with every change.

Apart from modularity, we can measure code complexity using lines of code as a proxy. A high-level comparison of model code sizes can be found in Table I. Even though this is not an apples-to-apples comparison, by virtue of the DSL toolchain, the Python version runs on a whole range of different hardware architectures, while the FORTRAN code only runs on multicore CPUs. While there are some differences between the codes (e.g., the FORTRAN version also supports hydrostatic modeling or nesting), there is an overall clear advantage to using Python, at 0.42× the code length.

B. Horizontal Regions

The cubed sphere grid on which FV3 runs contains correction terms that need to be applied at the corners and edges of each of the six tile faces in the grid. We thus extend GT4Py to support specializing stencils to sub-regions in the horizontal domain, such as corners and edges. For example, to specialize a computation on the lower edge in the second horizontal dimension, the syntax is as follows:

```
with computation(PARALLEL), interval(...):
  flux = dt2 * (velocity - velocity_c * cosa) / sina
  with horizontal(region[:, j_start]):
    flux = dt2 * velocity
```

If a stencil is distributed across multiple ranks in the horizontal plane, regions are also used to infer communication-related aspects. As only some of the regions read computed data from other ranks, the new DSL extension resolves which other ranks to synchronize with based on the ranges.

C. Halo exchange

The communication patterns we encounter in FV3 are point-to-point, which we realize with MPI. Each horizontal subdomain only needs to communicate a subset of its results on the boundary to the neighboring subdomains, often called a *halo*. Halo updates are slightly more complex on the cubed-sphere grid, as data must be transformed according to the orientation of the coordinate system of the adjoining faces of the cube. We thus design a halo updater object in Python that takes care of nonblocking communication, data packing, and transformation based on the pair of ranks.

D. Concessions

During porting, some concessions had to be made to create an equivalent model that can be statically compiled.

GT4Py does not support absolute indices (e.g., `a[I - i]`) and data-dependent horizontal offsets (e.g., `a[b[0], 0, 0]`). This led to increase in code size in two instances. First, in the aforementioned horizontal region computations, some code had to be specialized for each edge/corner. Second, there are modules that behave identically, except for the horizontal direction. For example, identical computations that involve either `a[1, 0]` or `a[0, 1]`. As there is no way to parametrize the dimension as a function argument, these modules had to be duplicated.

Another concession has to do with the GT4Py parallel model. Due to the abstraction of loops, some synchronization points were pre-determined and had to be worked around by splitting stencils into multiple functions.

Even though DaCe is flexible in compiling Python code, we had to limit the behavior of the code between stencils. In particular, we forgo dynamically-sized lists in performance-critical code and Python syntactic features such as introspection (e.g., live state `__dict__` updates). In order to ensure manageable size of the generated program, as a last consideration we explicitly mark loops to be (or not) unrolled.

VI. Hardware Mapping and Orchestration

With the FV3 dynamical core implemented as Python modules that invoke GT4Py stencils, in this work we implement a GT4Py backend that generates SDFGs (the DaCe IR). The goal is twofold: (a) GT4Py stencils can generate fast GPU code via data-centric optimizations; and (b) DaCe can parse the entire dynamical core as one unit, enabling global optimization.

A. From GT4Py to DaCe

Converting GT4Py stencils to SDFG occurs after GT4Py applies domain-specific optimizations (such as removing unnecessary computations) on the input code. Upon translation to an SDFG, each stencil computation is represented as a library node. These `StencilComputation` nodes are annotated with attributes to specify their *schedule* for hardware mapping. We define a stencil schedule as follows:

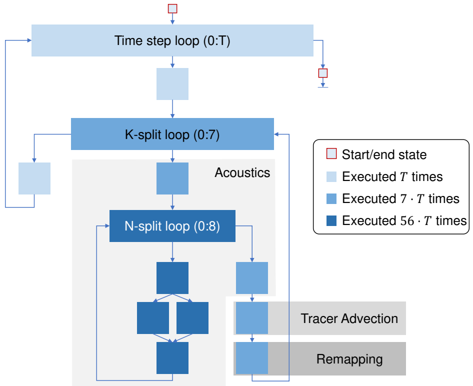


Fig. 6: Coarse-grain state machine for the configuration of the FV3 dynamical core used in our simulations.

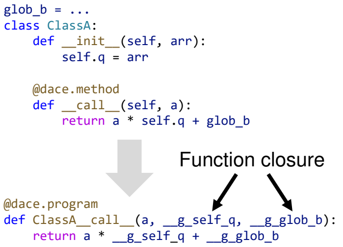
- Order of iteration, i.e., along which dimension grid points are accessed with unit stride.
- Tiling and tile sizes in each dimension.
- Target for expanded SDFG maps, e.g., GPU thread-block.
- Whether iteration in each dimension should be scheduled as a map or a loop. This allows to trade parallelism for better caching of accessed values.
- Fields for which data is retained in caches are specified alongside the kind of storage for those caches, such as shared memory or registers.
- Horizontal Regions can be implemented as separate maps (i.e., multiple kernels) with an iteration over the respective sub-domain or as a map over the full domain with code predicated on the index for individual statements.

The schedule thus defines a general optimization space for a single stencil. Based on the dependencies between data accesses in each node, only a subset of combinations of these settings is valid. For each node, we are able to generate a list of feasible options from which we make a preferred choice, which can be used for tuning. For some settings, default values are selected such that caches always reside in the fastest memory possible. For example, cached values that are accessed by a single thread will be held in registers, while those that require synchronization between threads are kept in shared memory.

When a Python function is parsed by DaCe, calls to GT4Py stencils are detected and insert the resulting library nodes into the graph. The representation can thus also be used to analyze and optimize stencils on an inter-procedural level, by considering subgraphs with multiple such nodes.

B. Orchestration

Full-program optimization is a key to striking a balance between performance and productivity. As the Python FV3 implementation defines and orders stencils by their physical meaning (e.g., diffusion or advection) rather than by opti-



```
glob_b = ...
class ClassA:
    def __init__(self, arr):
        self.q = arr

    @dace.method
    def __call__(self, a):
        return a * self.q + glob_b
```

```
@dace.program
def ClassA__call__(a, __g_self_q, __g_glob_b):
    return a * __g_self_q + __g_glob_b
```

Fig. 7: Closure resolution for a data-centric method.

mization opportunities, it is beneficial to consider optimizing transformations across multiple stencil invocations.

Constructing a single SDFG from the entire program creates several performance benefits. It eliminates the Python interpreter's overhead, enables optimizations across functions and translation units (e.g., choosing a data layout to persist across an entire section of the dynamical core, propagating constants into GPU kernels), and allows the DaCe framework to remove unused memory and allocate it outside of the critical path. Additionally, we can guide performance engineering by modeling the performance characteristics (compute and memory bounds) of the entire program.

When orchestrating a code of such magnitude, it is important to ensure productivity is not reduced. By keeping a connection to the Python interpreter via callbacks, our orchestration retains the capability to print out arrays, call external modules during execution (such as plotting libraries), validation, debugging with breakpoints, and use specialized methods for data I/O and boundary conditions.

In the Python FV3 implementation, apart from GT4Py stencils the code uses several design patterns. As mentioned, the code utilizes OOP via classes and fields, uses dictionaries, and includes control flow expressed as loops and branches (some depending on fields and dictionaries). Fig. 6 provides a schematic overview of FV3 as a state machine.

To make the FV3 Python code analyzable w.r.t. data movement, we implement a new Python preprocessor that can handle the aforementioned design patterns:

- **Constant Propagation**: The first step propagates constants forward (i.e., `i = 5; j = i + 1; b = a[j]` will become `b = a[6]`), performs loop unrolling, and dead code/branch elimination (namely, removing code that will definitely not be executed). This handles cases such as dictionary accesses in a loop (used, e.g., for tracers in FV3).
- **Closure Resolution**: Methods and functions that depend on external data are converted to free functions (Fig. 7). Resolving closures inlines class structures at preprocessing time, supporting Python OOP. With closures and constants resolved, a call-tree analysis detects and consolidates multiple instances of the same array object (e.g., used in different classes) to avoid data races.
- **Automatic Callbacks**: We extend DaCe to automatically generate callbacks for functions that cannot be parsed as C function pointers. Native Python containers are directly supported, mapping CuPy arrays to GPU pointers and NumPy to CPU. To avoid reordering callbacks during data-centric optimization, we add a `__pystate` dummy data container as an input and output for each callback, similarly to Calotoiu et al. [38].

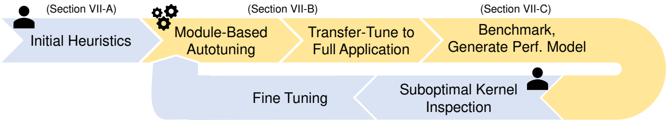


Fig. 8: Optimization pipeline cycle.

Working with large weather models, scalability becomes a requirement. After library node expansion, the SDFG of the orchestrated dynamical core comprises 26,689 dataflow nodes in 3,179 states, grouped into 4,241 unique GPU kernels (maps). Since the model contains loops, some of these kernels are invoked multiple times (≤ 56) under different settings. This necessitates a programmatic approach for optimization.

VII. Data-Centric Optimization

To accelerate the performance of the dynamical core, we realize a disciplined approach for optimization based on model-driven performance engineering [4]. Briefly, the application is divided into components on the critical path, and performance bounds (e.g., computational, I/O complexity) are computed on each component and the program as a whole. These serve as guides to how much could each component be optimized further, or hint that a global schedule change should be applied.

Performance engineers are always assumed to be present in a production environment, and this work creates a methodology that reduces the optimization cycle duration by automating as much of the pipeline as possible. As depicted in Fig. 8, the data-centric approach can be used to mechanize the modeling and tuning parts, which is enabled by the schedule-free declarative approach of GT4Py and the mutable data movement of the SDFG IR. Should the scientists modify the weather model's equations, consider a different model altogether, or upon using a new hardware architecture, the proposed pipeline can be easily reapplied.

The optimization pipeline is composed of four general steps: **initial heuristics** for scheduling stencils are determined based on estimates and local optimization. Subsequently, **autotuning** is performed on repeating subunits of the application (e.g., individual stencils, modules), and **transferred** to the full application. Automated performance bounds analyses then guide performance engineers to inspect and manually **fine tune** bottlenecks in the application, prompting the potential start of the next auto-tuning cycle. In the rest of this section, we provide a detailed account of applying this pipeline to FV3, annotating manual (👤) and automatic (⚙) optimizations.

A. Local optimization and initial heuristics

As a first step, the pipeline should optimize each stencil and allocated field locally using the infrastructure available in GT4Py and DaCe. In this work, we formalize the search space for local stencil optimization, and then automatically

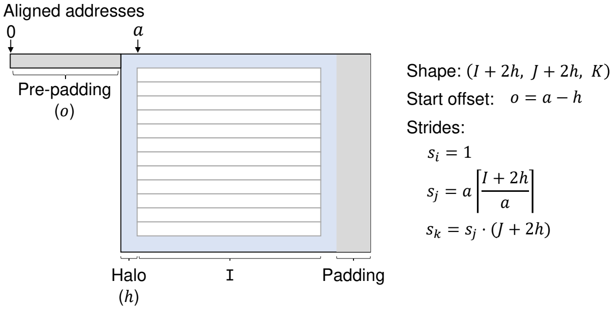
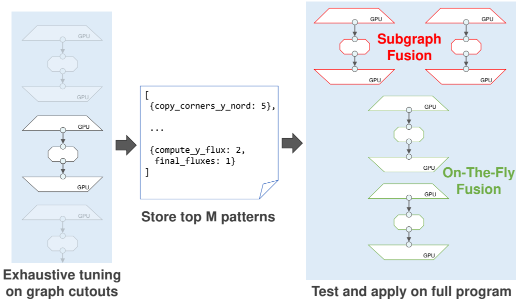

Aligned addresses
0
a
Pre-padding
(o)
Halo
(h)
I
Padding
Shape: $(I+2h,\ J+2h,\ K)$
Start offset: $o = a - h$
Strides:
$s_i = 1$
$s_j = a\left\lceil\frac{I+2h}{a}\right\rceil$
$s_k = s_j \cdot (J+2h)$

Fig. 9: Memory allocation scheme for desired alignment a.



```
[
 {copy_corners_y_nord: 5},

 ...

 {compute_y_flux: 2,
  final_fluxes: 1}
]
```

Fig. 10: Transfer tuning scheme.

search it on a representative horizontal stencil and vertical solver separately. The system then applies the resulting scheme *en masse* in the dynamical core, providing a better starting point over the default parameters.

There are four aspects to optimize: scheduling strategy for individual stencils, local storage (specifically for vertical solvers), memory allocation for fields, and computational layout (thread to work mapping).

1) ⚙ Scheduling: Optimizations that will yield a performance benefit are applied aggressively within GT4Py. Between statements in each stencil, the default fusion strategy combines consecutive intervals in forward and backward solvers into a single map (and GPU kernel), which allows to avoid flushing and re-initialization of cached values to and from global memory between loops. Subsequently, kernel fusion is applied on the thread level if no dependency between threads exists, removing unnecessary inter-thread synchronization. Lastly, consecutive operations in a single `computation` with dependencies between threads is fused by redundantly computing the results of the first operation (as can be seen in Fig. 5b).

2) ⚙ Local storage: The following transformations are applied to avoid load and store operations from or to global memory: (a) temporary fields that are only accessed in a single thread are replaced by local variables; (b) load operations for fields that are overwritten before being read are removed; and (c) values that are used in consecutive iterations of forward and backward solvers need only to be loaded from global memory on their first access, buffered locally on registers during iterations, and flushed back to global memory only when not needed again.

3) 👤 Memory allocation: We design a domain-specific optimization space for memory layout of stencil fields. Allocating the two- and three-dimensional fields used in the model can be parameterized by several knobs. While the shape is fixed, strides can affect layout and padding, whereas pre-padding can be applied to ensure certain elements are aligned. For our experiments, we use the scheme depicted in Fig. 9. FORTRAN data layout (`I`-contiguous) is used since it generates wide loads on the largest dimension (especially vertical solvers, which only access one value of `K` at a time). To create coalesced writes and reads of fields without offset, we apply pre-padding such that the first non-halo element is aligned, yielding up to 20 μs ($\sim$5%) of improvement on the tested stencil.

4) ⚙ Computational layout: Following an automated sweep of all valid layouts, the results dictate the following schedules. For horizontal stencils: `[Interval, Operation, K, J, I]` (last dimension corresponds to `threadIdx.x`). For vertical solvers, we use `[J, I, Interval, Operation, K]`. These results also correspond to the above data layout and improves coalesced access on the GPU.

B. ⚙ *Transfer Tuning*

Auto-tuning provides a convenient optimization framework, as it does not require a hand-crafted performance model. Unfortunately, exploring the configuration space of transformations for the entire dynamical core is infeasible for the size of FV3. A typical solution is to only consider a subset of the modules and tune them independently, e.g., the most frequent functions. However, certain *motifs* recur often in weather and climate codes and can be leveraged to further decrease tuning time. For instance, if fusing a 5-point stencil and a subsequent element-wise operation turns out to be a positive configuration, it will likely be a positive configuration in other modules, regardless of the actual function.

We propose a novel method to transfer knowledge across different regions of the code via extracting data movement patterns and reapplying them on other, untuned sections. Transfer Tuning happens in two phases (Fig. 10): in a first phase, the SDFG of the full program is divided into a set of "cutout" subgraphs, each of which is tuned individually. The best M configurations are translated into optimization patterns and tested on the whole graph in the second phase. The second phase ensures that found patterns are only applied if they also provide a local performance improvement on a match.

The second phase of transfer tuning requires a description of the patterns that can be searched for in the rest of the graph. We define a configuration by a set of names of the participating stencils and which transformations were applied, but other implementation-agnostic graph motifs could be used.

Case Study: Finite Volume Transport: We tune the `fv_tp_2d` module, which recurs often in FV3 (Fig. 2). There are 127 possible graph "cutouts" in the module, and optimization patterns found in it generalize to other parts of FV3. The maximum number of configurations for a cutout in this module is 48 and the total number of configurations is 1,272, which are searched exhaustively in the first phase.

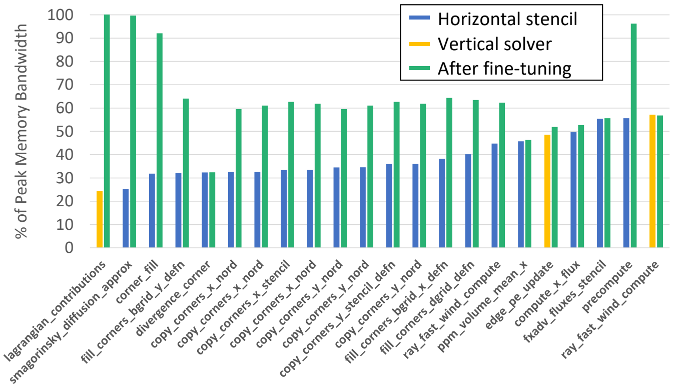


Fig. 11: Model-augmented kernel runtimes.

For the transfer, the best ($M = 2$) configurations of each cutout are considered on the target graph, which is the full dynamical core. In order to prune the space of matches, we only consider the first match for each pattern in each state, and only match the most performance-improving pattern. In total, 603 transformations were automatically found and transferred to the target FV3 graph. Without transfer tuning, the number of configurations to consider would be $\geq$ 30,302,185.

On a single node of the Piz Daint supercomputer, the runtime of the first phase of transfer tuning on the module takes 2:42 hours for pattern extraction. The second phase applied on FV3 takes 8:24 hours. Thus, auto-tuning the entire dynamical core can run in feasible time and drastically reduce the global optimization search space.

C. ⚙ Memory-bound analysis and 👤 fine-tuning

One of the advantages of using a data-centric IR is the capability to perform *automated* performance modeling. We use modeling, combined with runtime results, to create an overview for the performance engineer, breaking down which parts are most important to optimize further.

We write a simple, general script (17 lines of Python code) that computes the peak performance of each stencil, if it were memory bandwidth bound. The script avoids modeling L1 and L2 caches by considering every element of the field being accessed once, even if multiple threads access the same element. Since kernels execute under different configurations (due to array sizes or differently masked elements), performance engineers see the maximal reported runtime and largest modeled configuration for performance bounds analysis. To rank kernels by overall importance, the results are sorted by summarized runtimes grouped by kernel type.

In Fig. 11, we list the results of the worst-performing, most important kernels from our initial (i.e., first cycle) performance model. In subsequent cycles, as a result of further tuning, most of the shown kernels are above 60% peak memory bandwidth. Below, we showcase some of the manual fine-tuning efforts. The rest of the optimizations are outlined in Section X-A.

1) 👤 Case Study: Region Pruning: Fig. 11 shows that many of the kernels named `"*corner*"` under-perform based on the model. Upon performance engineer inspection, it was found that horizontal regions (Section V-B) are the root cause. Specifically, computations such as the following:

```
with horizontal(region[i_start - 1, j_start - 1]):
    q = sw_mult * q_corner[0, 1, 0]
```

would generate code that spawns many more threads than moved data elements, especially on ranks that do not require computation in the corners. As a result, many of the spawned GPU threads remain idle.

To resolve this, the performance engineers wrote a specialized DaCe transformation that detects such discrepancies via pattern-matching in GT4Py and (a) eliminates unused statements in regions based on MPI rank; and (b) generates smaller GPU kernels that do not spawn idle threads. With that, many of the corner/edge kernels run twice faster, but still not able to fully utilize the GPU memory bandwidth, likely due to the small data size processed.

2) 👤 Case Study: Smagorinsky Diffusion: As the second-slowest kernel in Fig. 11, Smagorinsky Diffusion demonstrates the ease of finding performance discrepancies with our model. The kernel appears to under-perform albeit not containing any specialized I/O complexity patterns nor excessive floating point operations. Upon manual inspection of its source code, the stencil uses the power operator extensively:

```
with computation(PARALLEL), interval(...):
  vort = dt * (delpc ** 2.0 + vort ** 2.0) ** 0.5
```

The resulting generated code contained `pow(delpc, 2.0)` and `pow(..., 0.5)`, which are general-purpose and thus highly inefficient. We thus created a transformation that converts powers of positive and negative integers, as well as 0.5, into multiplication loops and `sqrt` respectively. Following the transformation, the kernel runtime is reduced from 511.16 μs to 129.02 μs, with the automated model reporting 99.68% utilization. The runtime of other kernels improved as well, resulting in a 1.81% overall speedup for each time step.

VIII. Experimental Setup

In the following sections, we measure the performance of the Python FV3 implementation and estimate the efforts to apply the methodology to other models or hardware.

All experiments were conducted on the Swiss National Supercomputing Centre's Piz Daint supercomputer. The cluster contains 5,704 Cray XC50 nodes, each with an Intel Xeon E5-2690 v3 12-core CPU, NVIDIA Tesla P100 GPU (16 GB RAM), and 64 GB of host RAM. The nodes are connected via the Cray Aries interconnect. Each experiment is performed at least 10 times and the median result is reported.

We use Python 3.8.2, DaCe version 0.13, GT4Py revision `6588aa8`, fv3gfs-fortran revision `9030cf9`, and PAPI 6.0.0.9. Python FV3 was compiled with CUDA 11.2 and GCC 9.3.0, whereas the FORTRAN FV3 was compiled with Intel IFORT version 19.1.3.304. For all experiments, unless otherwise stated, we use a domain size of 192 × 192 horizontal grid-points per compute node and 80 vertical levels with double-precision elements. Using this setup on all of the nodes of Piz Daint would result in a global simulation of

TABLE II: Performance Analysis of Representative FV3 Modules

	Riemann Solver					Finite Volume Transport				
	FORTRAN		GT4Py+DaCe			FORTRAN		GT4Py+DaCe		
Domain Size (relative size)	Time [ms]	Scaling	Time [ms]	Scaling	Speedup	Time [ms]	Scaling	Time [ms]	Scaling	Speedup
128×128×80 (1x)	12.27	—	1.85	—	6.63×	3.41	—	1.81	—	1.88×
192×192×80 (2.25x)	27.94	2.28	3.86	2.08	7.25×	12.31	3.61	3.41	1.88	3.61×
256×256×80 (4x)	52.40	4.27	6.96	3.76	7.53×	35.79	10.49	5.67	3.13	6.31×
384×384×80 (9x)	121.80	9.92	15.31	8.26	7.96×	106.66	31.27	13.10	7.23	8.14×

1.5 km resolution. Our performance baseline is the optimized FORTRAN version of FV3, which is tuned for multicore CPUs. The FORTRAN version's configuration was tuned for production runs, at 6 ranks per node with 4 threads in each rank, thereby utilizing hyperthreading on the 12 physical cores.

IX. Performance Bounds

Following the model-driven performance engineering discipline, we begin by characterizing the workload. On hardware architectures where memory movement is orders of magnitude more expensive than computation, stencil programs will be memory-bound. This stems from the fact that operations in a stencil are typically proportional to the number of accesses. This is also the case in FV3. We use PAPI [40] to measure the FORTRAN dynamical core and observe that 40.15% of the executed instructions were load/store operations.

A. Memory bandwidth

For our target local domain size, we claim that FV3 is in particular a memory bandwidth-bound workload. To empirically prove this claim, as well as understand the expected speedups of GPU vs. CPU, we measure the peak and maximum attainable memory bandwidth of the two architectures.

We use the STREAM benchmark for the CPU [41] and the CUDA memory bandwidth test [42] for the GPU. The reported numbers for Piz Daint are 43.77 GB/s for the Haswell CPU, and 501.1 GB/s for the Pascal GPU. To verify that GT4Py and DaCe can achieve maximal bandwidth, we run a "copy stencil" (one input, one output) on the target domain size. We measure a CPU bandwidth of 40.99 GiB/s and a GPU memory bandwidth of 489.83 GiB/s. This indicates that the sizes are large enough to sustain the full bandwidth, as well as expect a maximum speedup of 11.45× for a memory-bound problem.

B. Vertical solvers: Riemann Solver

In the acoustic substep, the `riem_solver_c` module solves for the nonhydrostatic terms of vertical velocity and pressure perturbation [43], [44]. We use this module as a representative of vertical solvers, which typically do not perform well in the FORTRAN FV3 `K`-blocking schedule.

The semi-implicit method used for discretization is implemented using a tridiagonal solver, which is divided into three GT4Py stencils and scheduled as 22 GPU kernels. Table II (left) lists performance measured on several domain sizes, as well as the relative scaling w.r.t. the ratio of grid points.

The table exhibits two scaling trends. On the one hand, the FORTRAN version scales increasingly worse as the domain size grows. Scaling worse than the ideal scaling indicates that the CPU caches no longer suffice for larger domains. Moreover, the increase in gap between the slowdown and domain size suggests that the schedule is suboptimal. Different data layouts (e.g., `K`-contiguous) could mitigate this effect, but without a DSL approach that would constitute an intrusive code modification for every stencil that accesses those fields.

The second scaling trend can be seen in the data-centric Python version. We see that the slowdown is always smaller than the domain size scaling. This typically indicates that not enough parallelism is exposed on the smaller domain sizes (only 2D thread grids are used). This can also be confirmed by the gap decreasing as the domain size increases. Overall, for all sizes starting from the target domain size, the speedup is over 7.25× over the FORTRAN version. The speedup remains relatively stable, and is expected to increase further (albeit slightly) with larger domain sizes.

C. Horizontal stencils: Finite Volume Transport

An integral part of FV3 is the `fv_tp_2d` module, which is a subroutine to compute fluxes for horizontal finite volume transport [6], [45].

In FORTRAN, the module is designed to be two-dimensional and mass-conserving, i.e., each call operates on a horizontal slice of the transported scalar, and there are no dependencies between the vertical levels. To achieve high performance efficiency, vertical `K`-blocking is employed. This schedule and blocking strategy make this module challenging to gain speedups over, as we observe that CPU caches are highly utilized ($\sim$0.13% of the load/store instructions end up as L3 cache misses on our target domain size).

The runtimes in Table II (right) indicate that the FORTRAN version relies on CPU caches. With each domain size increase, the slowdown scales at a higher factor. However, the gap between slowdown and ideal scaling is decreasing, further indicating that we are outside of the cache capacity regime. We can also see that the GPU is again underutilized, with runtime scaling at a lower factor than the number of grid points. As the CPU starts to access off-chip memory, we see that the speedups increase towards the bandwidth ratio.

Another way to look at this performance behavior is as a hardware/software co-design guideline. An ideal hardware architecture for sequences of horizontal stencils would have the off-chip memory bandwidth and core count of a GPU, but could retain cached memory *across kernels*, benefiting from the inter-stencil memory reuse similarly to a CPU.

TABLE III: Dynamical Core Optimization

Cycle	Version	Date	Time [s]	Speedup
	FORTRAN	—	16.36	1.00×
	GT4Py + DaCe (Default)	Feb 15	10.87	1.50×
Cycle 1	Stencil schedule heuristics	Mar 13	5.56	2.94×
	Local caching	Mar 15	5.45	3.00×
	Optimize power operator	Mar 17	5.35	3.06×
	Split regions to multiple kernels	Mar 18	4.82	3.39×
Cycle 2	Lagrangian contrib. reschedule	Mar 25	4.82	3.40×
	Region pruning	Mar 31	4.77	3.43×
	Transfer Tuning	Mar 31	4.61	**3.55**×

X. Distributed Performance

For our test case we set the initial state of the model corresponding to a uniform zonal flow with a perturbation which evolves into a baroclinic instability [46]. This analytical test case enables generation of arbitrary domain sizes and a fast visual verification of the results.

A. Optimization pipeline

The smallest distributed run configuration uses 6 nodes, where each node takes a single face of the cubed sphere. In this configuration, each node computes all of the special computations required at tile edges and corners. We use this 6-node run as a case study for productivity and performance improvement over the course of optimization. The results and dates of each version are summarized in Table III.

The optimization consisted of two performance engineering cycles over a six week period (including tooling development) by four developers, where we follow the optimization cycle laid out in Fig. 8. Once the modeling/tuning tools were developed, and the search spaces were defined, most optimizations took days to perform. The cycle informed the developers on which parts of the models to focus, and the parametrization enabled modifying individual components with ease.

More specifically, as the default schedules were suboptimal, auto-tuning stencils locally to obtain heuristics for scheduling (Section VII-A4) yields a 1.96× speedup. As the cycle progresses with model-driven fine tuning, optimized kernels yield smaller but non-negligible returns. At the end of the second cycle, we run another tuning pass on the same day, transferring the results from `fv_tp_2d` to the rest of the dynamical core for a 3.47% speedup. Additional cycles could improve the performance further (albeit with diminished returns), or reapplied to other hardware as discussed in Section XI.

It is important to keep in mind that *all* performance engineering was accomplished without modifying the user-code, but by applying optimizations in the toolchain.

B. Performance at scale

We run global simulation experiments, scaling from 54 nodes (15.6 km grid spacing) to 2,400 nodes (2.28 km grid spacing) and keeping the same grid size per process. The time per timestep is plotted in Fig. 12, showing the weak scaling of the Python FV3 vs. the FORTRAN version, where shaded regions denote the nonparametric 95% confidence intervals.

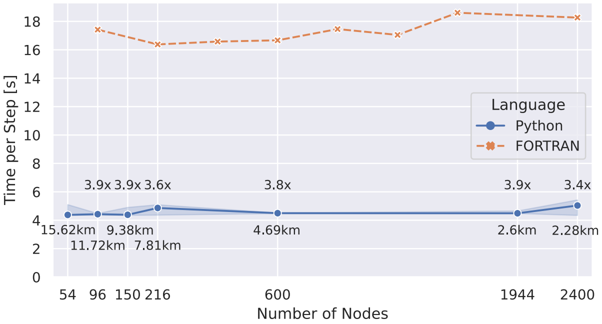


Fig. 12: Large-scale performance of FV3.

As expected, the speedups at scale are marginally higher (up to **3.92**×) than the ones obtained on the 6-node run. We observe a throughput of 0.11 simulated years per day (SYPD) for the 2.28 km simulation, and nearly perfect weak scaling (as per-node communication remains similar). This is all achieved with *object-oriented high-level declarative Python code*, which can be easily modified and further developed.

To demonstrate the portability of our approach, we also run the code unmodified on the JUWELS Booster supercomputer, which consists of multi-GPU nodes with NVIDIA Tesla A100 GPUs. With 54 ranks (14 nodes), we attain a performance of 1.93 seconds per timestep. It is 2.42× faster than the Piz Daint result, which is promising given that the memory bandwidth of A100 is 2.83× faster than the P100 GPU [47], [48].

XI. Productivity

The effort to port the dynamical core to Python took approximately two person-years. This includes not only the numerical operators, but also the initialization code, driver routines, and testing infrastructure. With the tooling now implemented, future models should be easier to port. Once the first version of the Python model validated against the reference FORTRAN model, the optimization efforts commenced.

Our approach for performance engineering is, by design, portable to different hardware architectures. However, re-optimization time is dependent on the target architecture type.

Table IV summarizes which elements of our methodology need to be repeated. *Compile* refers to the need to generate and recompile the code for a specific hardware target and is done with a single re-run of the model. *Tuning* refers to the transfer tuning approach described in Section VII-B. *Optimization cycles* refer to estimating the times the modeling-inspection-tuning cycle from Fig. 8 should be repeated, in order to achieve similar %peak performance. Lastly, *code generator* refers to writing a backend for DaCe, if there was no prior one.

While the P100 GPU was the target of this effort, performance on the A100 GPUs (Section X-B) was generated by just recompiling the code (i.e., without further tuning or an optimization cycle). When a different GPU architecture (e.g., AMD) is used, the heuristics and transformations would need to be re-tuned for hardware specifics such as warp size.

For CPUs, we know from FV3 that some of the GPU schedules are suboptimal for CPU, due to cache behavior and

TABLE IV: Workflow Porting Effort Estimation

Porting	Compile	(Transfer) Tuning	Optimization Cycles	Code Generator
NVIDIA GPU	✓		0	
Other GPU	✓	✓	0	
CPU	✓	✓	≈ 1–2	
Other chip	✓	✓	≥ 2	✓
New model, same GPU	✓	✓	≈ 1	

different instruction sets (e.g., ARM SVE). We thus estimate that another optimization cycle or two would be additionally necessary to ensure the best performance.

XII. Conclusion

We present a disciplined approach to optimizing weather and climate applications. Using FV3 as a case study, we demonstrate how taking a declarative approach to defining the underlying equations enables full freedom to optimize each stencil locally, and reschedule the model as a whole. Powering this flexible optimization is a data-centric representation, which allows performance engineers to automatically establish performance bounds and optimize computation and data layout, without modifying the source code.

The shown pipeline results in a successful porting of FV3, which is specifically designed and tuned for multi-core CPUs, to run on GPUs at scale, all while taking less than a half of the original code. The methodology presented here is applicable to any weather model and various hardware architectures, where it could yield similar, if not higher performance improvements.

Acknowledgment

This project received funding from the European Research Council under the European Union's Horizon 2020 programme (Project PSAP, No. 101002047); and EuroHPC-JU funding under grants DEEP-SEA, No. 955606 and MAELSTROM, No. 955513, with support from the Horizon 2020 programme. The authors also wish to acknowledge the support from the PASC program (Platform for Advanced Scientific Computing) for the DaceMI project. T.B.N. is supported by the Swiss National Science Foundation (Ambizione Project #185778). 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 acknowledge the contributions of Christopher Kung (NASA) for his contributions to the port of FV3 to GT4Py. Furthermore, we acknowledge the helpful discussions with Lucas Harris (GFDL) and Rusty Benson (GFDL) to understand the inner workings of FV3. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID s1053. We would like to thank the Allen Institute for Artificial Intelligence (AI2) and Vulcan Inc. for their financial support for part of the staff of this project.

References

[1] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär, "Reflecting on the goal and baseline for exascale computing: a roadmap based on weather and climate simulations," *Computing in Science & Engineering*, vol. 21, no. 1, pp. 30–41, 2018.

[2] B. N. Lawrence, M. Rezny, R. Budich, P. Bauer, J. Behrens, M. Carter, W. Deconinck, R. Ford, C. Maynard, S. Mullerworth, C. Osuna, A. Porter, K. Serradell, S. Valcke, N. Wedi, and S. Wilson, "Crossing the chasm: how to develop weather and climate models for next generation computers?" *Geoscientific Model Development*, vol. 11, no. 5, pp. 1799–1821, 2018. [Online]. Available: https://gmd.copernicus.org/articles/11/1799/2018/

[3] P. Bauer, P. D. Dueben, T. Hoefler, T. Quintino, T. C. Schulthess, and N. P. Wedi, "The digital revolution of earth-system science," *Nature Computational Science*, vol. 1, no. 2, pp. 104–113, 2021. [Online]. Available: https://doi.org/10.1038/s43588-021-00023-0

[4] T. Hoefler, W. Gropp, M. Snir, and W. Kramer, "Performance Modeling for Systematic Performance Tuning," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), SotP Session*, Nov. 2011.

[5] S.-J. Lin, "A "Vertically Lagrangian" Finite-Volume Dynamical Core for Global Models," *Monthly Weather Review*, vol. 132, no. 10, pp. 2293–2307, Oct. 2004, publisher: American Meteorological Society Section: Monthly Weather Review. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/132/10/1520-0493_2004_132_2293_avlfdc_2.0.co_2.xml

[6] W. M. Putman and S.-J. Lin, "Finite-volume transport on various cubed-sphere grids," *Journal of Computational Physics*, vol. 227, no. 1, pp. 55–78, Nov. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999107003105

[7] M. Ji and F. Toepfer, "Dynamical Core Evaluation Test Report for NOAA's Next Generation Global Prediction System (NGGPS)," *NOAA Institutional Repository*, 2016, publisher: United States. National Oceanic and Atmospheric Administration. [Online]. Available: https://repository.library.noaa.gov/view/noaa/18653

[8] L. Harris, L. Zhou, S.-J. Lin, J.-H. Chen, X. Chen, K. Gao, M. Morin, S. Rees, Y. Sun, M. Tong, B. Xiang, M. Bender, R. Benson, K.-Y. Cheng, S. Clark, O. D. Elbert, A. Hazelton, J. J. Huff, A. Kaltenbaugh, Z. Liang, T. Marchok, H. H. Shin, and W. Stern, "GFDL SHiELD: A Unified System for Weather-to-Seasonal Prediction," *Journal of Advances in Modeling Earth Systems*, vol. 12, no. 10, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1029/2020MS002223

[9] L. M. Harris and S.-J. Lin, "A Two-Way Nested Global-Regional Dynamical Core on the Cubed-Sphere Grid," *Monthly Weather Review*, vol. 141, no. 1, pp. 283–306, Jan. 2013, publisher: American Meteorological Society Section: Monthly Weather Review. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/141/1/mwr-d-11-00201.1.xml

[10] B. Stevens, M. Satoh, L. Auger, J. Biercamp, C. S. Bretherton, X. Chen, P. Düben, F. Judt, M. Khairoutdinov, D. Klocke, C. Kodama, L. Kornblueh, S.-J. Lin, P. Neumann, W. M. Putman, N. Röber, R. Shibuya, B. Vanniere, P. L. Vidale, N. Wedi, and L. Zhou, "DYAMOND: the DYnamics of the Atmospheric general circulation Modeled On Non-hydrostatic Domains," *Progress in Earth and Planetary Science*, vol. 6, no. 1, p. 61, Dec. 2019. [Online]. Available: https://progearthplanetsci.springeropen.com/articles/10.1186/s40645-019-0304-z

[11] T. Palmer and B. Stevens, "The scientific challenge of understanding and estimating climate change," *Proceedings of the National Academy of Sciences*, vol. 116, no. 49, pp. 24 390–24 395, Dec. 2019, publisher: Proceedings of the National Academy of Sciences. [Online]. Available: https://www.pnas.org/doi/10.1073/pnas.1906691116

[12] W. Putman, K. Datta, A. Oloso, and T. Clune, "Finite volume advection kernel optimization on state-of-the-art heterogeneous architectures," in *UCAR Software Engineering Assembly*. University Corporation for Atmospheric Research, 2012.

[13] M. Govett, J. Middlecoff, Y. Yu, D. Fiorino, J. Rosinski, and L. Stringer, "Parallelization of the fv3 model for gpu and mic processors," in *Third Symposium on High Performance Computing for Weather, Water, and Climate*. American Meteorological Society, 2017.

[14] NVIDIA Corporation, "CUDA FORTRAN," https://developer.nvidia.com/cuda-fortran, 2022.

[15] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn, “Workflows are the new applications: Challenges in performance, portability, and productivity,” in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 57–69.

[16] V. Clement, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, and W. Sawyer, “The claw dsl: Abstractions for performance portable weather and climate models,” in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3218176.3218226

[17] C. Osuna, T. Wicky, F. Thuering, T. Hoefler, and O. Fuhrer, “Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications,” *Supercomputing Frontiers and Innovations*, vol. 7, no. 2, p. 79–97, Jul. 2020. [Online]. Available: https://superfri.org/index.php/superfri/article/view/314

[18] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, “Domain-specific multi-level ir rewriting for gpu,” 2020. [Online]. Available: https://arxiv.org/abs/2005.13014

[19] T. Schneider, S. Lan, A. Stuart, and J. Teixeira, “Earth system modeling 2.0: A blueprint for models that learn from observations and targeted high-resolution simulations,” *Geophysical Research Letters*, vol. 44, no. 24, pp. 12–396, 2017.

[20] S. Adams, R. Ford, M. Hambley, J. Hobson, I. Kavčič, C. Maynard, T. Melvin, E. Müller, S. Mullerworth, A. Porter, M. Rezny, B. Shipway, and R. Wong, “Lfric: Meeting the challenges of scalability and performance portability in weather and climate models,” *Journal of Parallel and Distributed Computing*, vol. 132, pp. 383–396, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518305306

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] M. Abadi, “Tensorflow: learning functions at scale,” in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 2016, pp. 1–1.

[23] S. Hoyer and J. Hamman, “xarray: Nd labeled arrays and datasets in python,” *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[24] Met Office, *Cartopy: a cartographic python library with a matplotlib interface*, Exeter, Devon, 2010 - 2015. [Online]. Available: http://scitools.org.uk/cartopy

[25] J. McGibbon, N. D. Brenowitz, M. Cheeseman, S. K. Clark, J. P. Dahm, E. C. Davis, O. D. Elbert, R. C. George, L. M. Harris, B. Henn *et al.*, “fv3gfs-wrapper: a python wrapper of the fv3gfs atmospheric model,” *Geoscientific Model Development*, vol. 14, no. 7, pp. 4401–4409, 2021.

[26] S. Partee, M. Ellis, A. Rigazzi, S. Bachman, G. Marques, A. Shao, and B. Robbins, “Using machine learning at scale in HPC simulations with smartsim: An application to ocean climate modeling,” *CoRR*, vol. abs/2104.09355, 2021. [Online]. Available: https://arxiv.org/abs/2104.09355

[27] N. Brenowitz, “call_py_fort,” https://github.com/nbren12/call_py_fort, 2022.

[28] J. M. Monteiro, J. McGibbon, and R. Caballero, “sympl (v. 0.4. 0) and climt (v. 0.15. 3)–towards a flexible framework for building model hierarchies in python,” *Geoscientific Model Development*, vol. 11, no. 9, pp. 3781–3794, 2018.

[29] M. Curcic, “A parallel fortran framework for neural networks and deep learning,” *CoRR*, vol. abs/1902.06714, 2019. [Online]. Available: http://arxiv.org/abs/1902.06714

[30] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, and P. Baldi, “A fortran-keras deep learning bridge for scientific computing,” *Scientific Programming*, vol. 2020, 2020.

[31] K. G. Pressel, C. M. Kaul, T. Schneider, Z. Tan, and S. Mishra, “Large-eddy simulation in an anelastic framework with closed water and entropy balances,” *Journal of Advances in Modeling Earth Systems*, vol. 7, no. 3, pp. 1425–1456, 2015.

[32] J.-B. Lin, “qtcm 0.1. 2: A python implementation of the neelin-zeng quasi-equilibrium tropical circulation model,” *Geoscientific Model Development*, vol. 2, no. 1, pp. 1–11, 2009.

[33] B. E. J. Rose, “climalab,” https://github.com/brian-rose/climlab, 2022.

[34] GT4Py Authors, “GT4Py: GridTools for Python,” https://github.com/GridTools/gt4py, 2022.

[35] A. Afanasyev, M. Bianco, L. Mosimann, C. Osuna, F. Thaler, H. Vogt, O. Fuhrer, J. VandeVondele, and T. C. Schulthess, “Gridtools: A framework for portable weather and climate applications,” *SoftwareX*, vol. 15, p. 100707, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711021000522

[36] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, “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*, ser. SC ’19, 2019.

[37] A. N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler, “Productivity, portability, performance: Data-centric Python,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476176

[38] A. Calotoiu, T. Ben-Nun, G. Kwasniewski, J. de Fine Licht, T. Schneider, P. Schaad, and T. Hoefler, “Lifting C semantics for dataflow optimization,” in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS ’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3524059.3532389

[39] O. Rausch, T. Ben-Nun, N. Dryden, A. Ivanov, S. Li, and T. Hoefler, “A data-centric optimization framework for machine learning,” in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS ’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3524059.3532364

[40] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with papi-c,” in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[41] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[42] NVIDIA Corporation, “CUDA Toolkit Official Samples,” https://github.com/NVIDIA/cuda-samples, 2022.

[43] X. Chen, N. Andronova, B. V. Leer, J. E. Penner, J. P. Boyd, C. Jablonowski, and S.-J. Lin, “A Control-Volume Model of the Compressible Euler Equations with a Vertical Lagrangian Coordinate,” *Monthly Weather Review*, vol. 141, no. 7, pp. 2526–2544, Jul. 2013, publisher: American Meteorological Society Section: Monthly Weather Review. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/141/7/mwr-d-12-00129.1.xml

[44] X. Chen, S. Lin, and L. M. Harris, “Towards an Unstaggered Finite-Volume Dynamical Core With a Fast Riemann Solver: 1-D Linearized Analysis of Dissipation, Dispersion, and Noise Control,” *Journal of Advances in Modeling Earth Systems*, vol. 10, no. 9, pp. 2333–2356, Sep. 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1029/2018MS001361

[45] S.-J. Lin and R. B. Rood, “Multidimensional Flux-Form Semi-Lagrangian Transport Schemes,” *Monthly Weather Review*, vol. 124, no. 9, pp. 2046–2070, Sep. 1996, publisher: American Meteorological Society Section: Monthly Weather Review. [Online]. Available: https://journals.ametsoc.org/view/journals/mwre/124/9/1520-0493_1996_124_2046_mffslt_2_0_co_2.xml

[46] P. A. Ullrich, T. Melvin, C. Jablonowski, and A. Staniforth, “A proposed baroclinic wave test case for deep-and shallow-atmosphere dynamical cores,” *Quarterly Journal of the Royal Meteorological Society*, vol. 140, no. 682, pp. 1590–1602, 2014.

[47] NVIDIA Corporation, “Tesla P100 Datasheet,” https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf, 2016.

[48] ——, “Tesla A100 Datasheet,” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf, 2021.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

The experiments run in the paper measure the performance of the orchestrated, data-centric Python FV3 implementation.

Most experiments were conducted on the Swiss National Supercomputing Centre's Piz Daint supercomputer, and one experiment on the Juelich JUWELS Booster supercomputer. Piz Daint contains Cray XC50 nodes with Intel Xeon E5-2690 v3 CPUs, NVIDIA Tesla P100 GPU (16 GB RAM), and 64 GB of host RAM; whereas JUWELS Booster contains AMD EPYC 7402 CPUs, four NVIDIA Tesla A100 GPUs (40 GB RAM) per node, and 512 GB of host RAM.

Our performance baseline is the optimized FORTRAN version of FV3, which is tuned for multi-core CPUs. The code was obtained from the fv3gfs-fortran repository (https://github.com/ai2cm/fv3gfs-fortran/tree/9030cf9c9241d087e8fcdef71ab9458aee980609). FORTRAN FV3 was compiled with Intel IFORT version 19.1.3.304, and was tuned for production runs on Piz Daint, at 6 ranks per node with 4 threads in each rank, thereby utilizing hyperthreading on the 12 physical cores. We use PAPI 6.0.0.9 to obtain CPU performance counters for benchmarking.

For our novel Python FV3 implementation, we use Python 3.8.2. The main package used is fv3core (https://github.com/ai2cm/fv3core/tree/SC22), which is based on modified versions of DaCe 0.13 (https://github.com/spcl/dace/tree/FV3v2) and GT4Py (https://github.com/tbennun/gt4py/tree/SC22-AE). The code is subsequently compiled to GPUs using CUDA 11.2, GCC 9.3.0, and CMake 3.17.0. On Piz Daint, the MPI version used is the GPU-aware Cray MPI (version 7.7.18).

For all end-to-end experiments, we use a domain size of 192x192 horizontal grid-points per Piz Daint node and 80 vertical levels with double-precision elements.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: https://doi.org/10.5281/zenodo.6973549
Artifact name: Artifacts, Code, and Docker Container

Artifact 2

Persistent ID: https://github.com/spcl/dace/tree/FV3v2
Artifact name: DaCe Persistent URL

Artifact 3

Persistent ID: https://github.com/tbennun/gt4py/tree/SC22-AE
Artifact name: GT4Py Persistent URL

Artifact 4

Persistent ID: https://github.com/ai2cm/fv3gfs-fortran/tree/9030cf9c9241d087e8fcdef71ab9458aee980609
Artifact name: Baseline FORTRAN Version and Docker Container

Artifact 5

Persistent ID: https://github.com/ai2cm/fv3core/tree/SC22
Artifact name: Python FV3 Persistent URL

Reproduction of the artifact with container: The artifact provides persistent GitHub links (as submodules) to specific git tags used for the benchmarks. These specific revisions are also built into the Dockerfile and published image.

There are two options to create the Docker container:

An image is available via Docker Hub: "docker pull tbennun/fv3-sc22:latest" (or ‘sarus pull‘ on Piz Daint)

The Dockerfile and all associated scripts can be built from the files in Artifact 1. Use the following command: "docker build -t <some tag> -f ./artifact.Dockerfile ." The NVIDIA Docker runtime (nvidia-docker) is required to expose the GPU to the container.

To run the workload, our large-scale experiment requires at least 6 ranks (each with its own GPU). Each rank will start up its own container, as shown in the following example using the SLURM job scheduler: "srun -N 6 nvidia-docker run tbennun/fv3-sc22:latest bash /runner.sh"

If on Piz Daint (where the container was tested), Sarus can be used to connect the container to the native MPI implementation and GPU: "srun -N 6 sarus run –mpi tbennun/fv3-sc22:latest bash /runner.sh"

We also provide a smaller example that can be run on a single node (provided that the GPU supports multiple processes), which can be run with: "nvidia-docker run tbennun/fv3-sc22:latest bash /runner_local.sh"

In order to run the baseline FORTRAN version, refer to the README containing Docker creation instructions in https://github.com/ai2cm/fv3gfs-fortran

The benchmark input folders provided here are compatible with the FORTRAN version.

For more information, see https://github.com/fv3ad/sc22-artifact