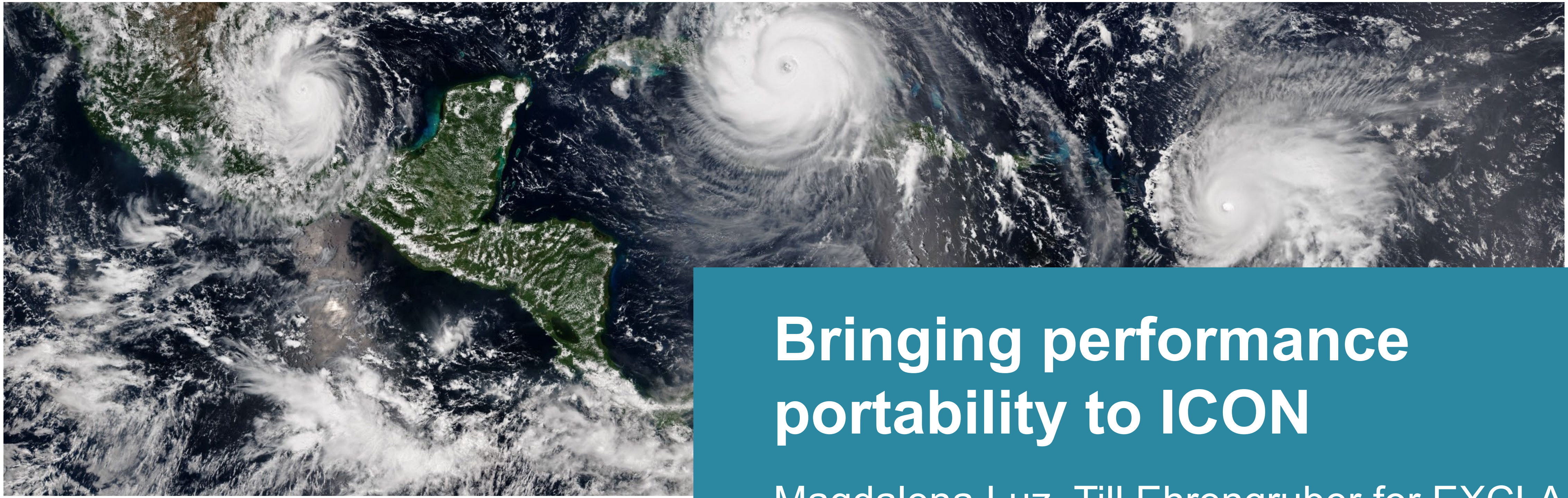
ETH zürich

EXCLAIM



Bringing performance portability to ICON

Magdalena Luz, Till Ehrengruber for EXCLAIM

18.09.2025 / 21st ECMWF workshop on high performance computing in meteorology

C2SM Center for Climate Systems Modeling

CSCS

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Home Affairs FDHA
Federal Office of Meteorology and Climatology MeteoSwiss

MeteoSwiss

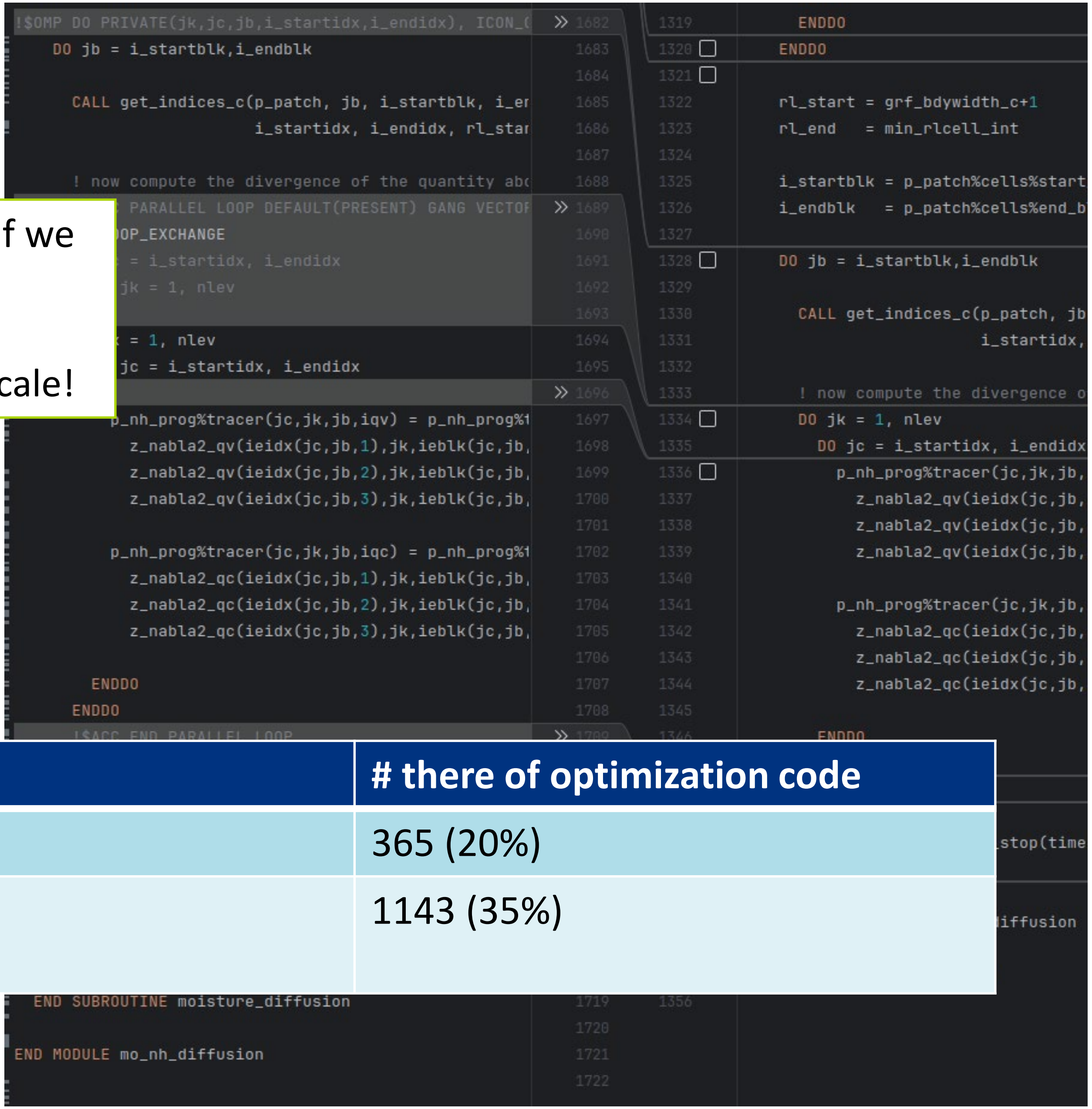
SDSC

Empa
Materials Science and Technology

Classical ICON Code

What if you only had the physics in the source code? If we could hide away all the clutter.

New paradigm – and new statements... this does not scale!

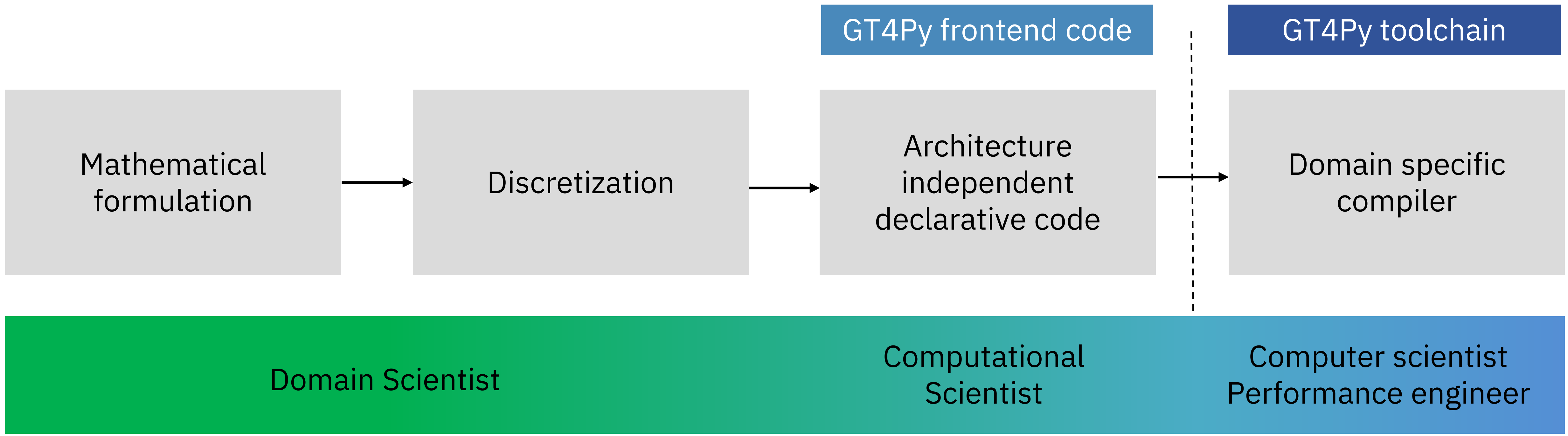


Module	# lines	# there of optimization code
Diffusion	1720	365 (20%)
Dynamical Core (without advection)	3170	1143 (35%)

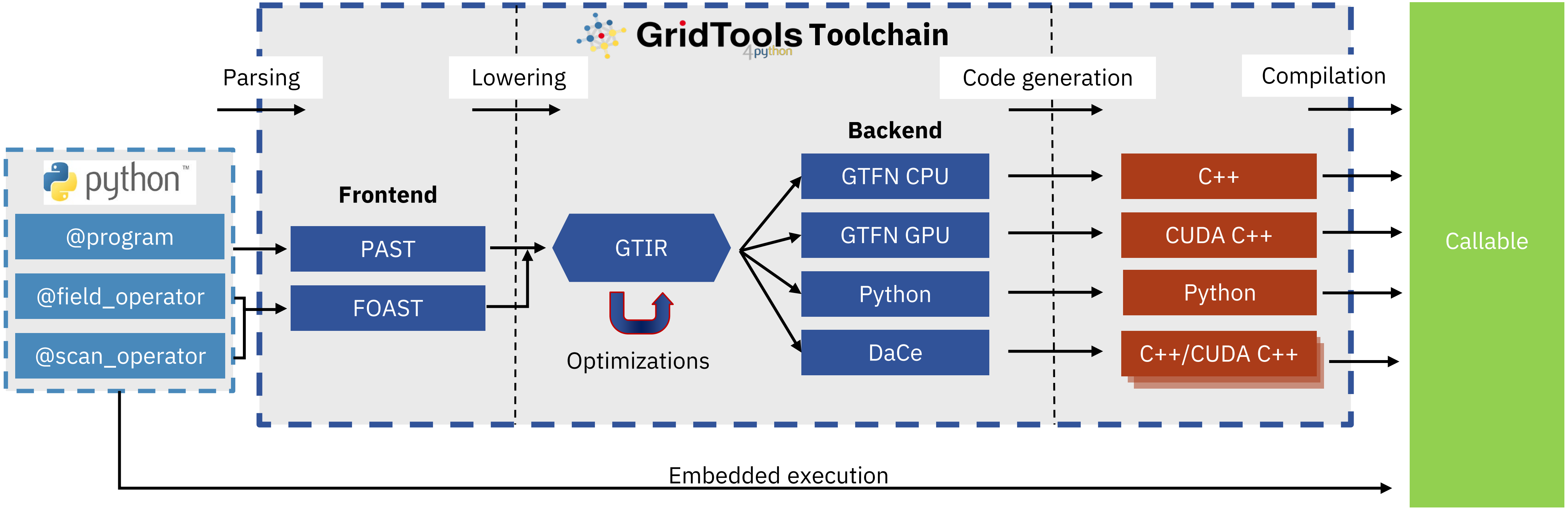
XCLAIM

Bring performance portability to an existing model

DSL brings separation of concerns:



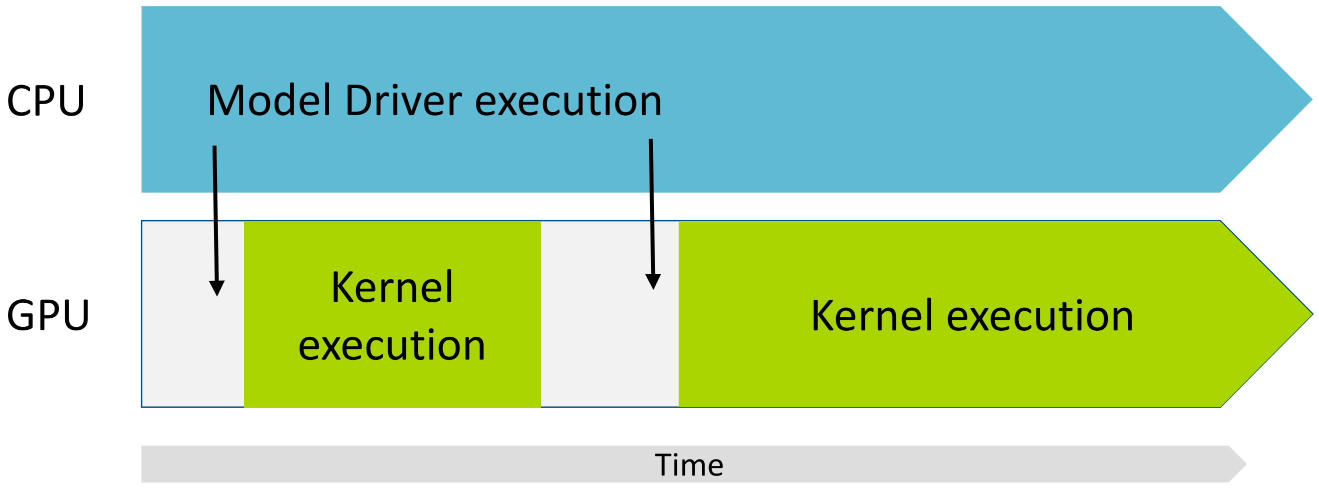
GT4Py Toolchain



EXCLAIM

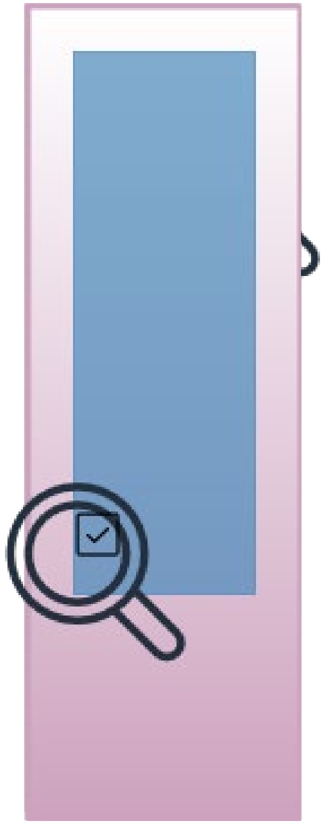
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Asynchronous execution



- Driver continues execution while computations happen on the GPU (or other CPU threads in the future).
- With some care Python overhead is already low, asynchronous execution hides the rest.
- Exception: call overhead
  - Already small even with many field operators (< 5%)
  - Negligible with large field operators
    - Fewer calls -> less overhead
    - Goal: one field operator per timestep

"Inside-out" Approach



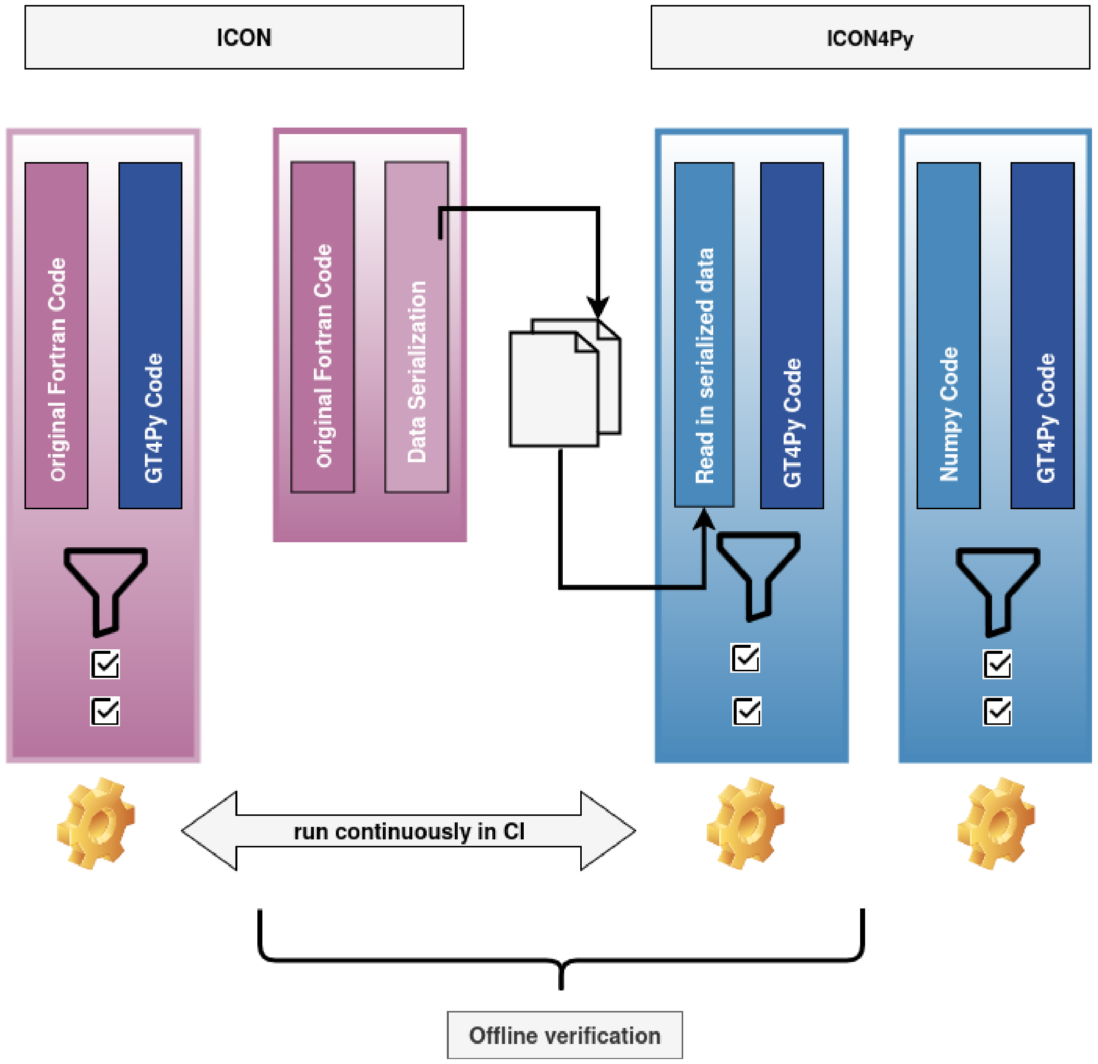
```python
class Diffusion:
    """Class that configures diffusion and does one diffusion step."""

    def __init__(
        self,
        grid: icon_grid.IconGrid,
        config: DiffusionConfig,
        params: DiffusionParams,
        vertical_grid: v_grid.VerticalGrid,
        metric_state: diffusion_states.DiffusionMetricState,
        interpolation_state: diffusion_states.DiffusionInterpolationState,
        geometrical_factors: grid_states.GeometryParams,
        backend: gtx_typing.Backend | None,
        exchange: decomposition.ExchangeRuntime | None = None,
    ):
    ...

    def run(self,
      diagnostic_state: diffusion_states.DiffusionDiagnosticState,
      prognostic_state: prognostics.PrognosticState,
       dtime: float,
    ):
    ...
```

```
ield[Dims[CellDim, K],float]]]:
```

Continuouesly test ... !!

Testing Strategies

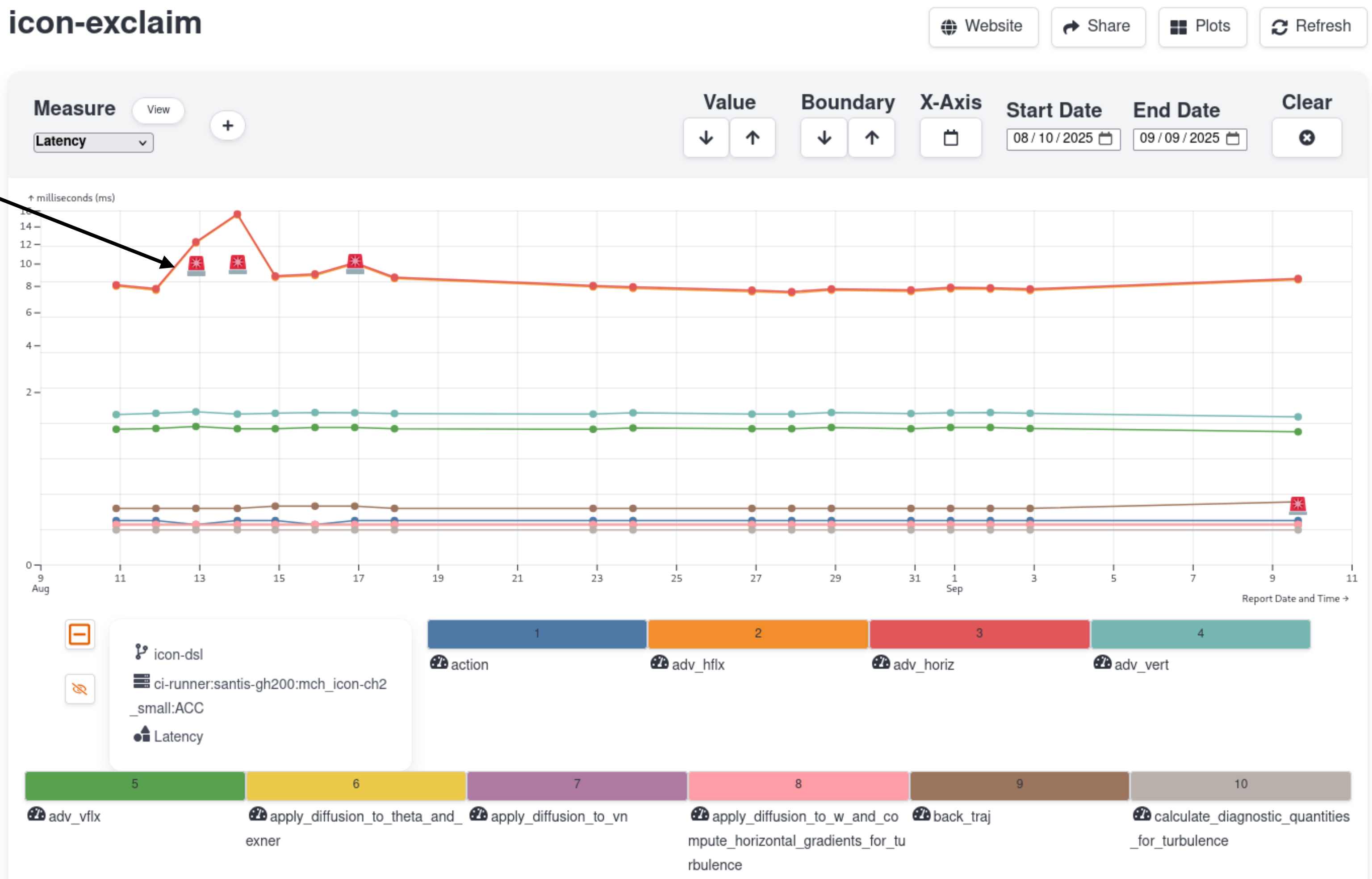
- Simplistic Testing: validate against numpy implementation – very accessible but unreliable!
  - Simple stencils
  - Combined stencils
- Verify inside ICON – run Fortran and GT4Py code
  - only components, stencil integration has been removed
- Hybrid: serialize data from ICON, run verification inside ICON4Py
  - Combined stencils
  - Components



XCLAIM

Continuous Benchmarking

Alert on performance degradation



https://bencher.dev/

Why choose such an approach?

Advantages

- Continuously **validate**
- Keep a version of the model running at all time
- Tackle complexity
- Build **trust** – in your own work, in the community

Disadvantages

- Restrictions in how to re-organize code internally and re-think the design
  - You must keep the functionality at the interface consistent
- Overhead – several integration modes along the way... to keep the model running

Blueline - Python dynamical core in FORTRAN

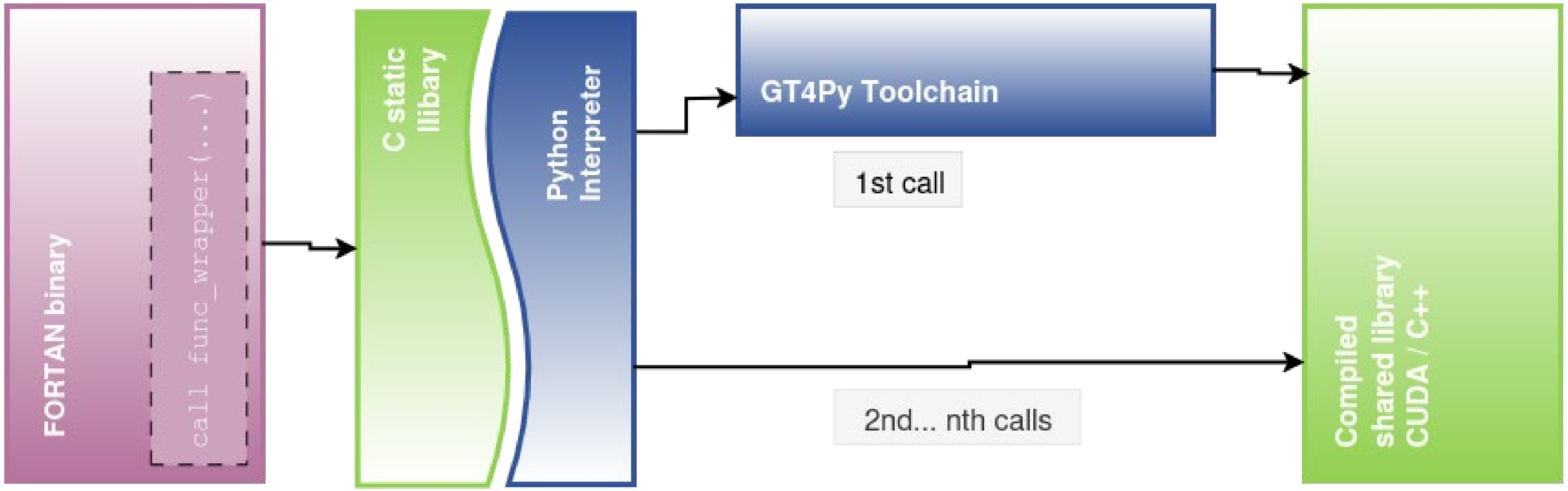
Run performance critical model components from the original FORTRAN code: `py2fgen` cli tool

1. Decorate function with `@py2fgen.export` and provide information on the function's

```
@py2fgen.export(
    param_descriptors={
        "scalar": py2fgen.ScalarParamDescriptor(py2fgen.FLOAT64),
        "array": py2fgen.ArrayParamDescriptor(
            rank=2, dtype=py2fgen.FLOAT64, memory_space=py2fgen.MAYBE_DEVICE, is_optional=False
        ),
    }
)
def foo(scalar: float, array: np.ndarray): ...
```

2. Run `py2fgen` cli tool - generates a Fortran interface and a (private) shared C library containing your python function

Blueline – Runtime view



- Uses Python [embedding feature of CFFI] under the hood
- Python interpreter is started only once and kept alive during runtime
- GT4Py tool chain triggered at first call

Greenline – Full Python model

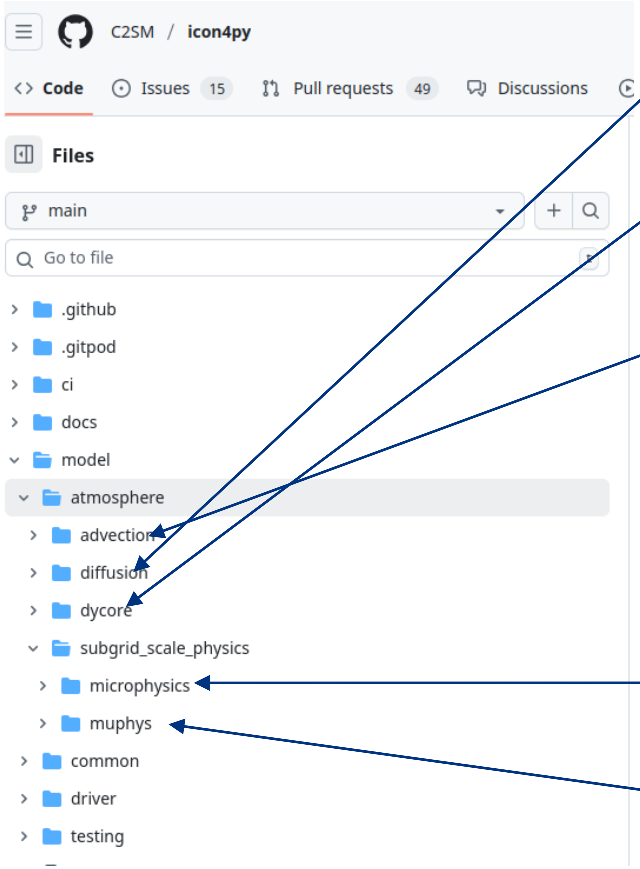
Re-implement the ICON model in Python.

- **Accessibility:** Python has a very rich ecosystem especially for scientific application – modeling code in Python allows to bridge the gap between model runs and analysis.

- **Interoperability:** Python is the language of the AI revolution, we need to make physical models interoperable with AI models. GT4Py is very close to be fully compatible with tools like JAX.

- **Productivity:** Performance portability taken one step further – develop & debug on your laptop, or in a jupyter notebook, run on HPC Cluster at scale

Where are we? - Physics components



Dynamical core

Diffusion

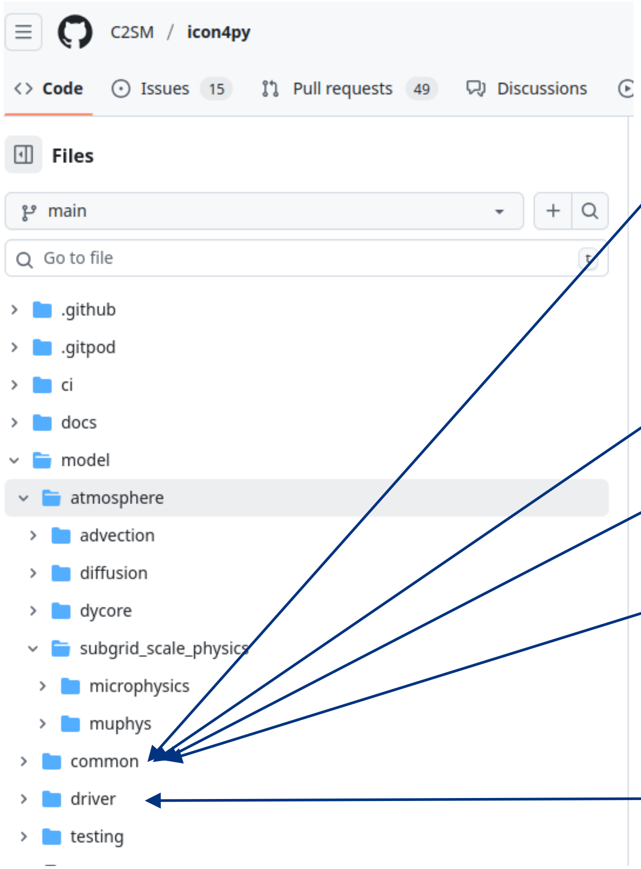
Tracer Advection

	vertical	horizontal
Scheme	Upwind 1st order PPM 3d order	Linear 2nd order
Limiter	Semi monotonic	Positive definite

ICON microphysics

Mu-phys scheme (MPI-M)

Where are we? - Model infrastructure



Grid Topology&Geometry
- Read topology
- Compute geometrical quantities

Basic common functionality
- Dimension definitions
- Allocation function
- Backend configuration

Precomputed static coefficients for horizontal and vertical interpolation

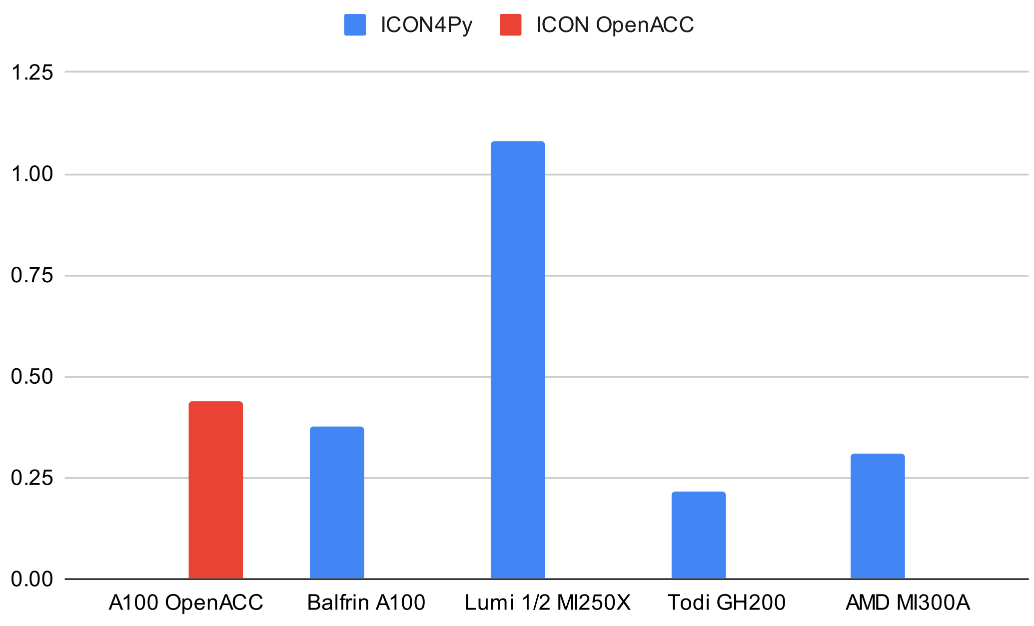
Decomposition
- Can do halo exchange on distributed grids.
- Distribution

Simple driver for atmospheric test cases
- Jablonoswki Williamson
- Gauss3d

Warm bubble

- Replace with more flexible, configurable driver (next)
- Enhance torus support

Greenline runs across systems



- Jablonowski Williamson test case on 20km resolution, single GPU runs
- Pure Python model ("EXCLAIM greenline")
- Python overhead from control flow is hidden by async scheduled kernels on GPU
- Took us ~1 day to get it run on AMD GPUs

EXCLAIM

Thank you!

https://github.com/c2sm/icon4py
https://github.com/GridTools/gt4py

M. Bianco[1], N. Buchendorfer[3], J.Canton[2], Y. Chen[2], O. Chia Rui[2], T. Ehrengruber[1], N. Farabullini[2], R. Häuselmann[1], D. Hupp[3], A. Jocksch[1], P. Kardos[2], J. Jucker[2], S. Kellerhals[2], C. Kotsalos[1], M. Luz[2], I. Magkanaris[1], C. Müller[3], P. Müller[1], E. G. Paredes[1] , E. Paone[1], W.Sawyer[1], M.Simberg[1], D. Strassmann[2], H. Vogt[1]

[1] Swiss National Supercomputing Center, CSCS
[2] Center for Climate Systems Modeling, ETH Zurich
[3] Federal Office of Meteorology and Climatology, MeteoSwiss