Pace

A GPU-Enabled Implementation of FV3GFS using GT4Py

Oliver Elbert, Johann Dahm, Eddie Davis, Florian Deconinck, Rhea George, Jeremy McGibbon, Tobias Wicky, Elynn Wu, Christopher Kung, Tal Ben-Nun, Lucas Harris, Linus Groner, and Oliver Fuhrer

The Pace Model

FV3 dynamical core, GFDL Cloud Microphysics v2 in Cartesian GT4Py

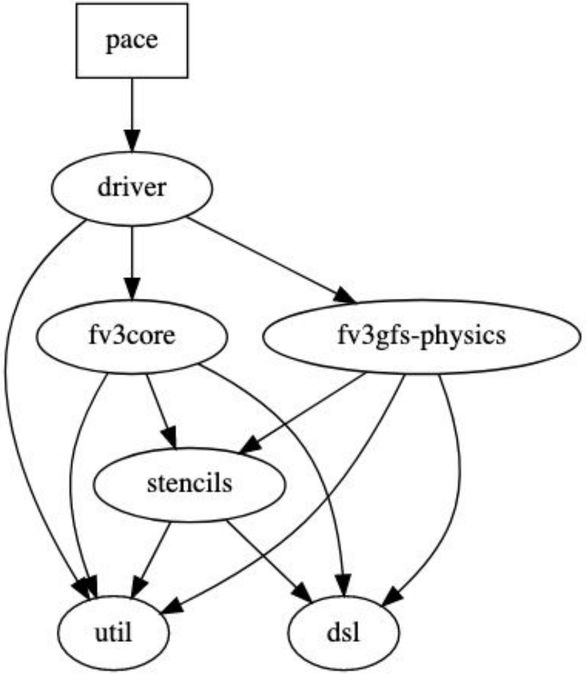
Contains infrastructure needed to run simulations

V3 of microphysics nearly finished, other GFS physics ported not optimized or integrated yet

VULCAN → Ai2 → NOAA

https://github.com/NOAA-GFDL/pace

Dahm et al.
https://gmd.copernicus.org/articles/16/2719/2023/

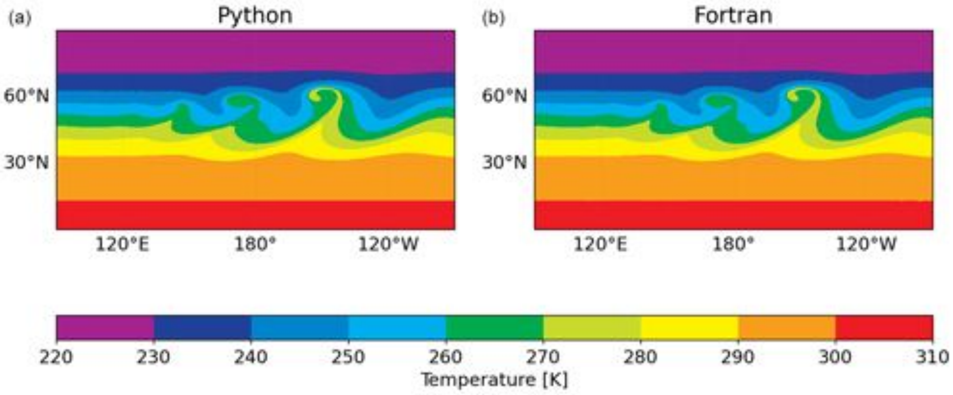


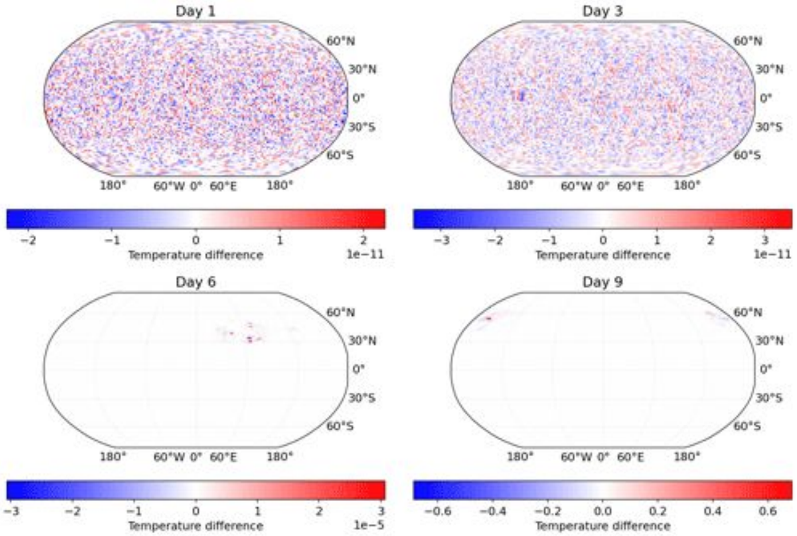
Comparing to Fortran

Moist baroclinic instability integrated for 9 days

Results match fairly well given arithmetic changes

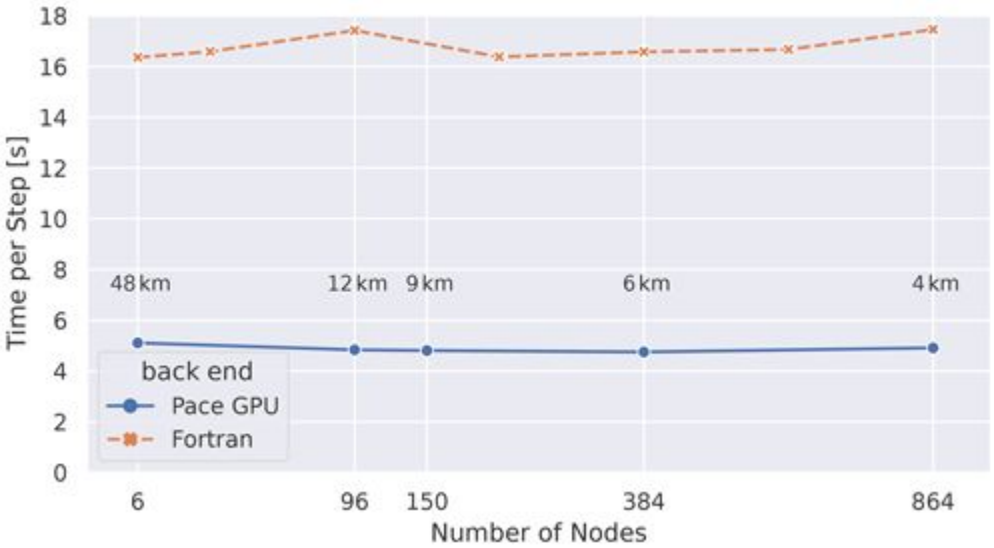
Plotted: 850 mbar temperature





Pace Performance

~3.6x speedup over Fortran on P100 GPUs, extra factor of ~2.4 on A100s



Ben-Nun et al: https://arxiv.org/pdf/2205.04148.pdf

CPU optimization coming soon

So…

What have we learned?

Lessons Learned So Far

1. **Can replicate Fortran model in a DSL**

Lessons Learned So Far

1. Can replicate Fortran model in a DSL
2. **GPU performance boost**

DSL approach

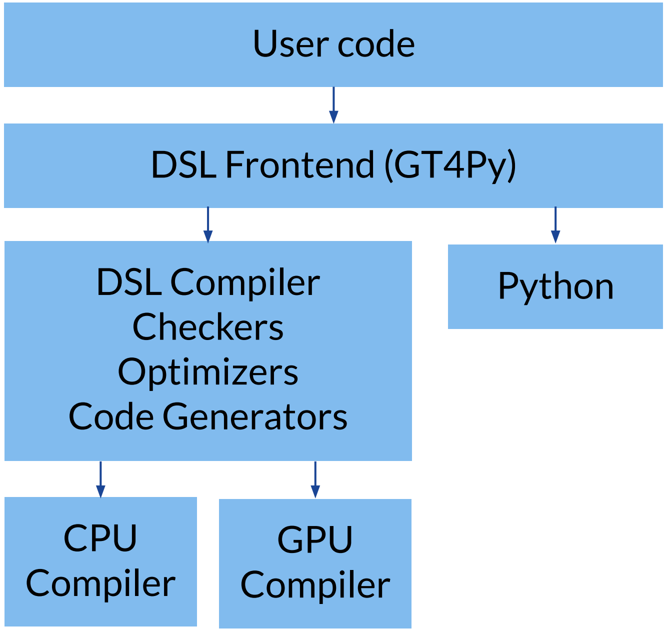
Leverage frontend/backend distinction!

Model development can be easy with readable, clean frontend Python

Portable code extremely helpful during GPU transition

Performance engineering details more separated from modeling

GridTools 4python



User code

DSL Frontend (GT4Py)

DSL Compiler
Checkers
Optimizers
Code Generators

Python

CPU Compiler

GPU Compiler

Lessons Learned So Far

1. Can replicate Fortran model in a DSL
2. GPU performance boost
3. **DSL paradigm is good**

FV3 Specifics

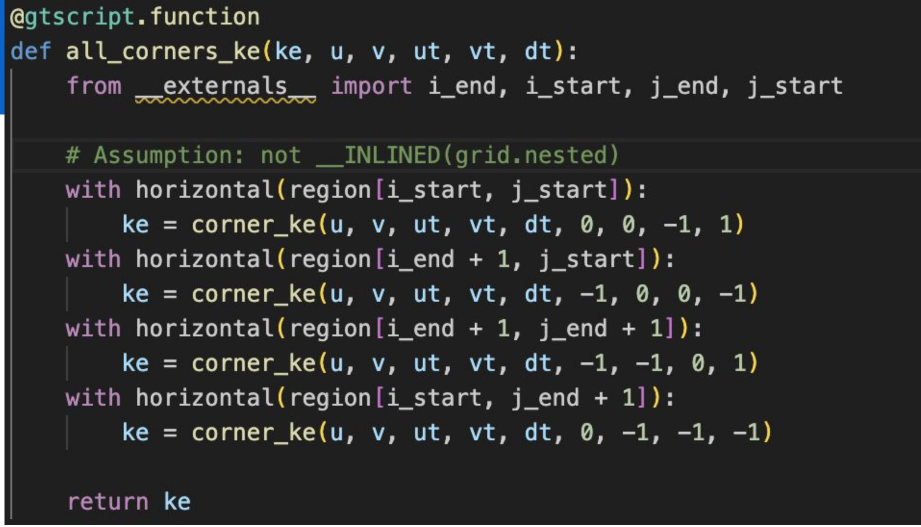
Extremely efficient Fortran dycore

Used extensively in NOAA and partner models (SHiELD, AM4, GEOS, GFS, HAFS…)

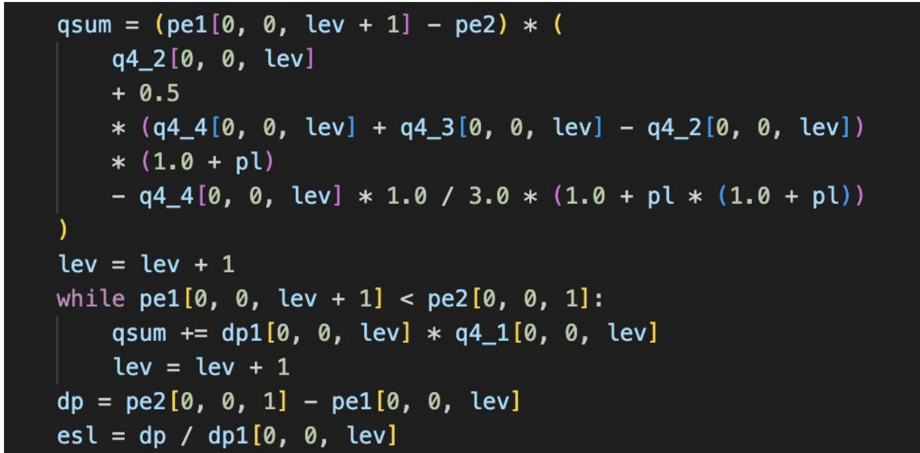
Finite volume dynamics on cubed-sphere C-D grid discretization

Special computations to account for tile edge/corner geometry

Lagrangian vertical coordinate regularly remapped to Eulerian coordinates



```
@gtscript.function
def all_corners_ke(ke, u, v, ut, vt, dt):
    from __externals__ import i_end, i_start, j_end, j_start

    # Assumption: not __INLINED(grid.nested)
    with horizontal(region[i_start, j_start]):
        ke = corner_ke(u, v, ut, vt, dt, 0, 0, -1, 1)
    with horizontal(region[i_end + 1, j_start]):
        ke = corner_ke(u, v, ut, vt, dt, -1, 0, 0, -1)
    with horizontal(region[i_end + 1, j_end + 1]):
        ke = corner_ke(u, v, ut, vt, dt, -1, -1, 0, 1)
    with horizontal(region[i_start, j_end + 1]):
        ke = corner_ke(u, v, ut, vt, dt, 0, -1, -1, -1)

    return ke
```



```
qsum = (pe1[0, 0, lev + 1] - pe2) * (
    q4_2[0, 0, lev]
    + 0.5
    * (q4_4[0, 0, lev] + q4_3[0, 0, lev] - q4_2[0, 0, lev])
    * (1.0 + pl)
    - q4_4[0, 0, lev] * 1.0 / 3.0 * (1.0 + pl * (1.0 + pl))
)
lev = lev + 1
while pe1[0, 0, lev + 1] < pe2[0, 0, 1]:
    qsum += dp1[0, 0, lev] * q4_1[0, 0, lev]
    lev = lev + 1
dp = pe2[0, 0, 1] - pe1[0, 0, lev]
esl = dp / dp1[0, 0, lev]
```

Object Orientation

Most stencils live inside classes

- **Preserves temporary storages**
- **Split init/compile time from runtime**
- **Simple organization**

`__init__` **creates an object of the class, handles stencil compilation, etc.**

`__call__` **means objects are called like functions**

```python
class XPiecewiseParabolic:
    """
    Fortran name is xppm
    """

    def __init__(
        self,
        stencil_factory: StencilFactory,
        dxa,
        grid_type: int,
        iord,
        origin: Index3D,
        domain: Index3D,
    ):
        assert grid_type < 3
        self._dxa = dxa
        ax_offsets = stencil_factory.grid_indexing.axis_offsets(origin, domain)
        self._compute_flux_stencil = stencil_factory.from_origin_domain(
            func=compute_x_flux,
            externals={
                "iord": iord,
                "mord": abs(iord),
                "xt_minmax": True,
                "i_start": ax_offsets["i_start"],
                "i_end": ax_offsets["i_end"],
            },
            origin=origin,
            domain=domain,
        )

    def __call__(
        self,
        q_in: FloatField,
        c: FloatField,
        q_mean_advected_through_x_interface: FloatField,
    ):
        """
        Args:
            q_in (in): scalar to be integrated
            c (in): Courant number (u*dt/dx) in x-direction defined on x-interfaces,
                indicates the fraction of the adjacent grid cell which will be
                advected through the interface in one timestep
            q_mean_advected_through_x_interface (out): defined on x-interfaces.
                mean value of scalar within the segment of gridcell to be advected
                through that interface in one timestep, in units of q_in
        """
        self._compute_flux_stencil(
            q_in, c, self._dxa, q_mean_advected_through_x_interface
        )
```

Lessons Learned So Far

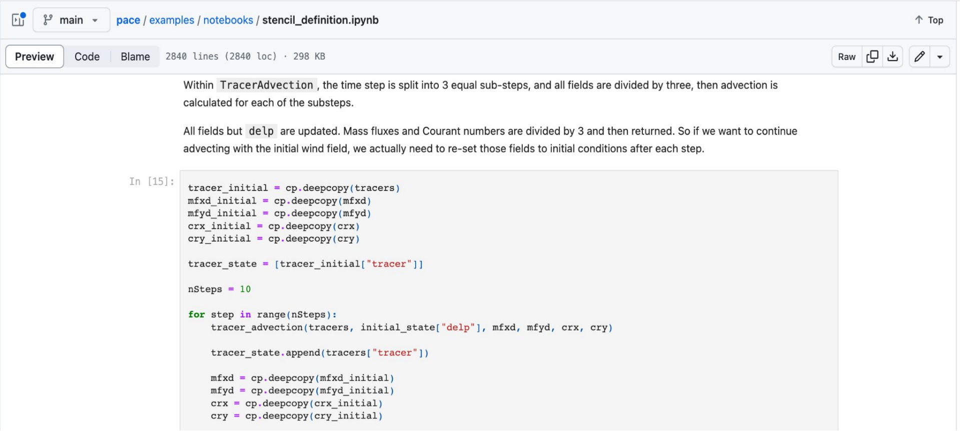
1. Can replicate Fortran model in a DSL
2. GPU performance boost
3. DSL paradigm is good
4. **Still need communication between frontend modeling and backend engineering**

Driving Adoption

Excitement about using Jupyter notebooks for model development

New tests and powerful Python debugging

More attractive as features increase and team grows…

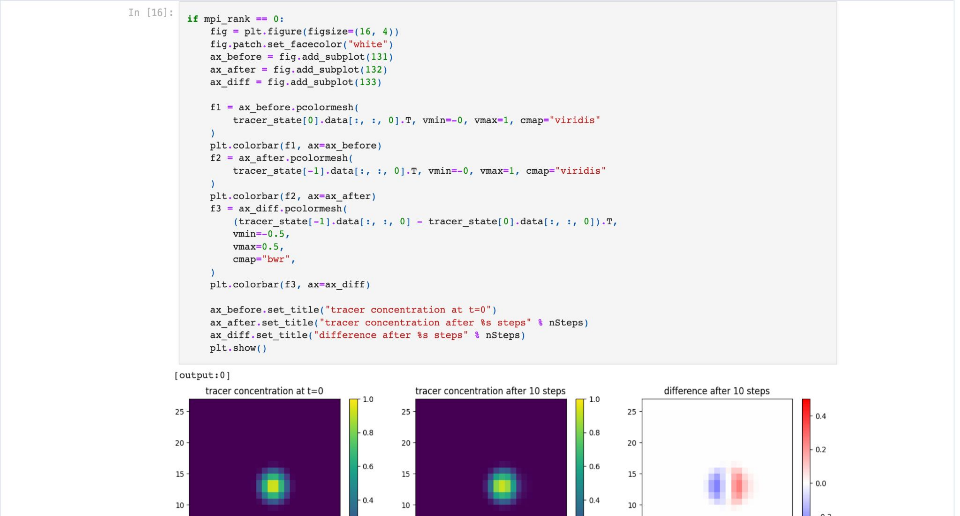


Within `TracerAdvection`, the time step is split into 3 equal sub-steps, and all fields are divided by three, then advection is calculated for each of the substeps.

All fields but `delp` are updated. Mass fluxes and Courant numbers are divided by 3 and then returned. So if we want to continue advecting with the initial wind field, we actually need to re-set those fields to initial conditions after each step.

In [15]:

```
tracer_initial = cp.deepcopy(tracers)
mfxd_initial = cp.deepcopy(mfxd)
mfyd_initial = cp.deepcopy(mfyd)
crx_initial = cp.deepcopy(crx)
cry_initial = cp.deepcopy(cry)

tracer_state = [tracer_initial["tracer"]]

nSteps = 10

for step in range(nSteps):
    tracer_advection(tracers, initial_state["delp"], mfxd, mfyd, crx, cry)

    tracer_state.append(tracers["tracer"])

    mfxd = cp.deepcopy(mfxd_initial)
    mfyd = cp.deepcopy(mfyd_initial)
    crx = cp.deepcopy(crx_initial)
    cry = cp.deepcopy(cry_initial)
```



In [16]:

```
if mpi_rank == 0:
    fig = plt.figure(figsize=(16, 4))
    fig.patch.set_facecolor("white")
    ax_before = fig.add_subplot(131)
    ax_after = fig.add_subplot(132)
    ax_diff = fig.add_subplot(133)

    f1 = ax_before.pcolormesh(
        tracer_state[0].data[:, :, 0].T, vmin=-0, vmax=1, cmap="viridis"
    )
    plt.colorbar(f1, ax=ax_before)
    f2 = ax_after.pcolormesh(
        tracer_state[-1].data[:, :, 0].T, vmin=-0, vmax=1, cmap="viridis"
    )
    plt.colorbar(f2, ax=ax_after)
    f3 = ax_diff.pcolormesh(
        (tracer_state[-1].data[:, :, 0] - tracer_state[0].data[:, :, 0]).T,
        vmin=-0.5,
        vmax=0.5,
        cmap="bwr",
    )
    plt.colorbar(f3, ax=ax_diff)

    ax_before.set_title("tracer concentration at t=0")
    ax_after.set_title("tracer concentration after %s steps" % nSteps)
    ax_diff.set_title("difference after %s steps" % nSteps)
    plt.show()
```

```
[output:0]
```

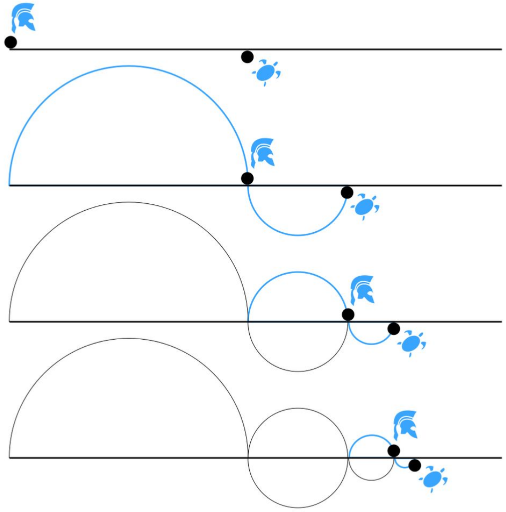
Driving Adoption

Minimum Useful Model

Add capabilities modelers want, meanwhile modelers keep developing Fortran

What capabilities allow for quickest use in research/forecasting/teaching?

- **RCE on doubly-periodic domain**
- **Dycore wrapper for Fortran model runs**



https://en.m.wikipedia.org/wiki/File:Zeno_Achilles_Paradox.png

Lessons Learned So Far

1. Can replicate Fortran model in a DSL
2. GPU performance boost
3. DSL paradigm is good
4. Still need communication between frontend modeling and backend engineering
5. **Performance isn’t enough**
    - **Need to identify critical features for adoption**

Next Steps

More physics

JAX backend for ML and DA applications

Research applications (RCE, LES, TC)

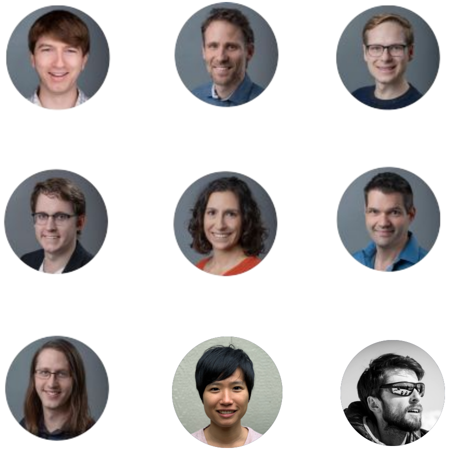
Incorporate into broader GFDL infrastructure

Growing collaboration and community

Image credit: Kun Gao

Thank you!

Former AI2 DSL team



FV3 Team, Modeling Systems Division

Collaborators

Ai2

CSCS

ETH

W
UNIVERSITY *of* WASHINGTON

NASA

MeteoSwiss