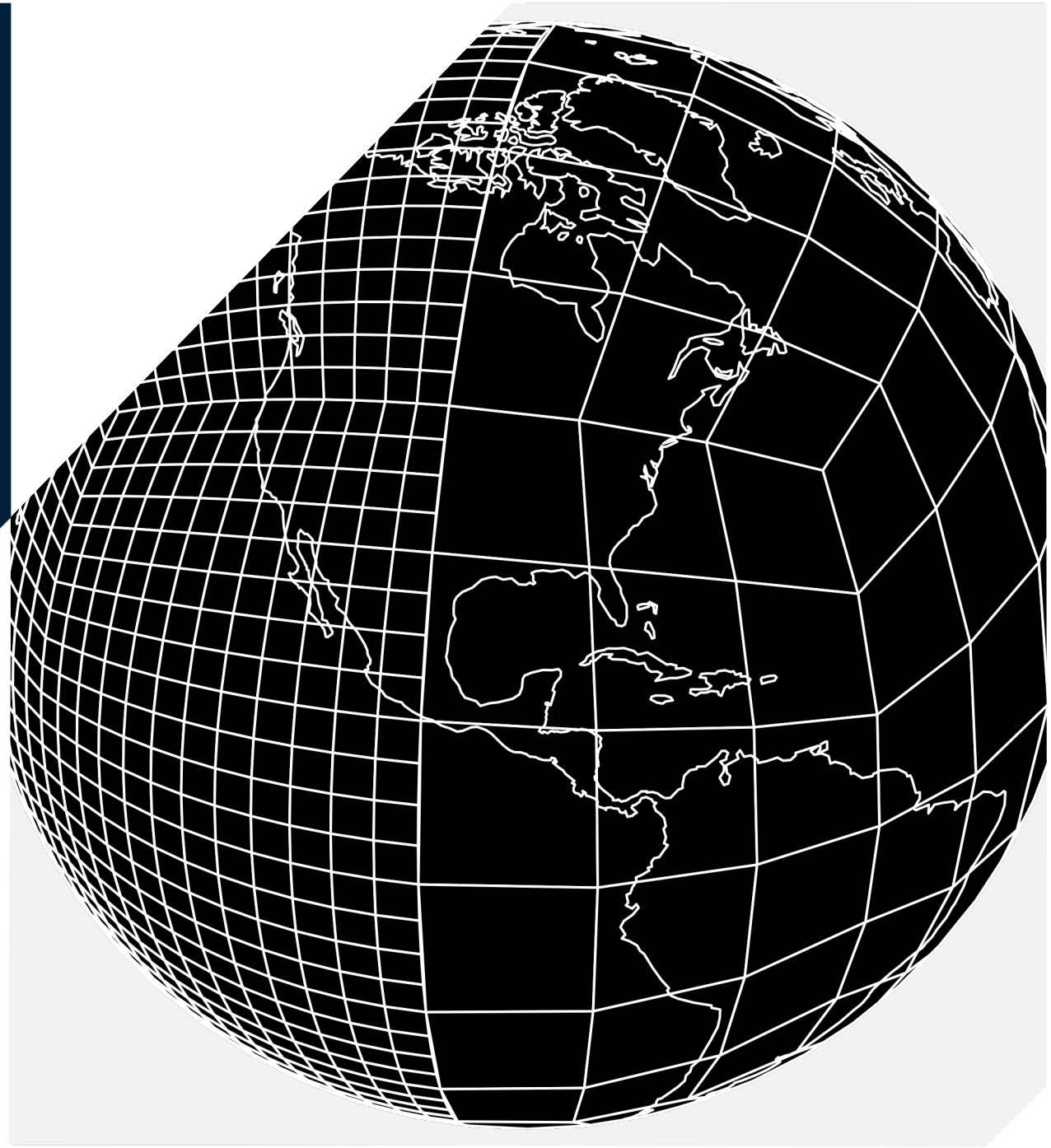
2020 ENES WORKSHOP

Compiler Toolchain for Scalable Weather and Climate Simulation using FV3 on GPUs

Johann Dahm <johannd@vulcan.com>, Eddie Davis, Tobias Wicky, Mark Cheeseman, Oliver Elbert, Oli Fuhrer, Rhea George, Jeremy McGibbon

Session 2: Performance Portability
28 May, 2020

VULCAN



Vulcan Climate Modeling

Vulcan is Paul Allen's company and engine for philanthropic efforts

Mission: Reduce uncertainties in model output based by machine learning and supercomputers

- ML: Learn parameterizations of physics models
- DSL: Enable execution on latest tech

VCM is partnered with GFDL in Princeton

Located in downtown Seattle

Part of the Impact Team – "Tech for Good"

Startup-like culture of software engineers and machine-learning scientists

Goal: Improve a climate model by enabling global storm-resolving atmospheric simulations on modern supercomputers

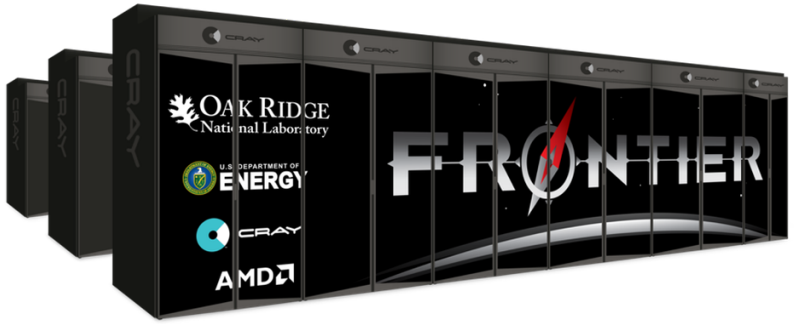
Multidecadal simulations and ensembles are still prohibitively expensive

But weather, seasonal, annual or possibly decadal timescales are within reach on large supercomputers

Push the envelope of what's possible!

Produce GSRM data for ML training

- **Year 1:** Rewritten dynamical core
- **Year 2:** Rewritten atmospheric model and demonstration run on supercomputer



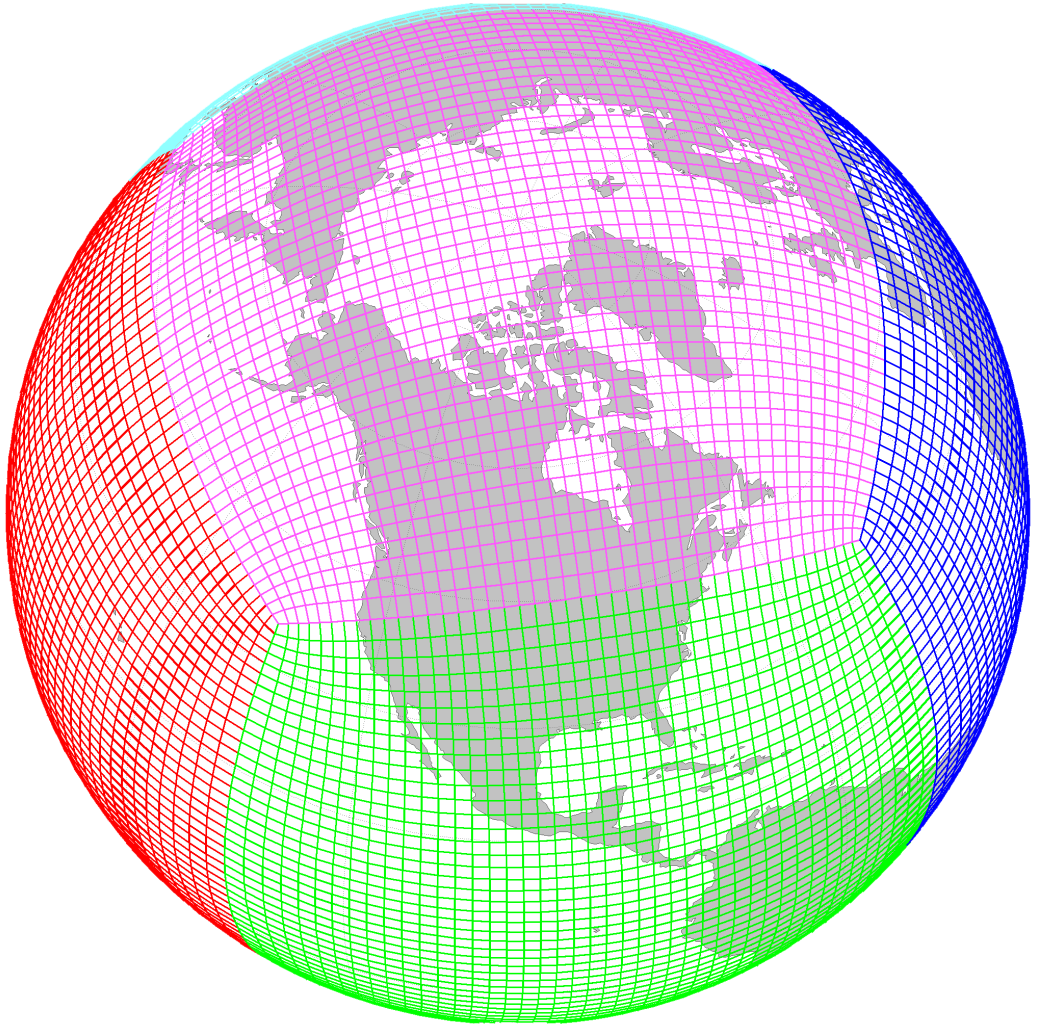
FV3GFS: Cubed-Sphere Grid

FV3GFS is solved using a structured grid on each of the six cube faces

MeteoSwiss and CSCS have ported the COSMO dycore regional model using the GridTools C++ framework and embedded DSL

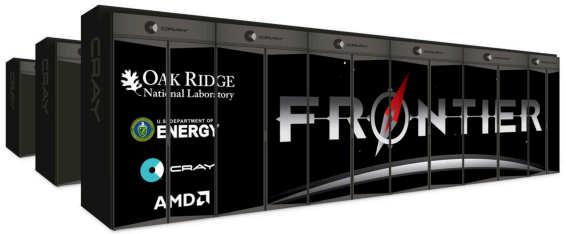
`github.com/GridTools/gridtools`

Leverage the underlying tools requires adapting tools to deal with edges and corners (similar as boundary conditions of regional models) and other new motifs



Development Roadmap

Fortran Code



Baseline Performance
(How to measure progress?)

Regression / Unit Testing
(Make sure nothing breaks!)

Refactored Code
(Domain-specific language in Python)

DSL Frontend (gt4py)
(Translate user code for compiler)

DSL Compiler (dawn)
(Generate efficient code)

Standard Compiler
(CPUs, GPUs, ...)

Demonstration Simulation
(What can we do now that was not possible before?)

Baseline Performance

Gaea (GFDL)
Cray XC40
2272 nodes (2 Broadwell CPUs)

Status
- FV3 ran at 200km, 13km, 3km
- No container support

Piz Daint (CSCS)
Cray XC50 5320 nodes
(Haswell CPU + P100 GPU)

Status
- FV3 ran at 200km, 13km, 3km
- Container support enabled

Google Cloud
Single VM (Skylake CPUs)

Status
- FV3 ran at 200km, 13km
- Container support enabled

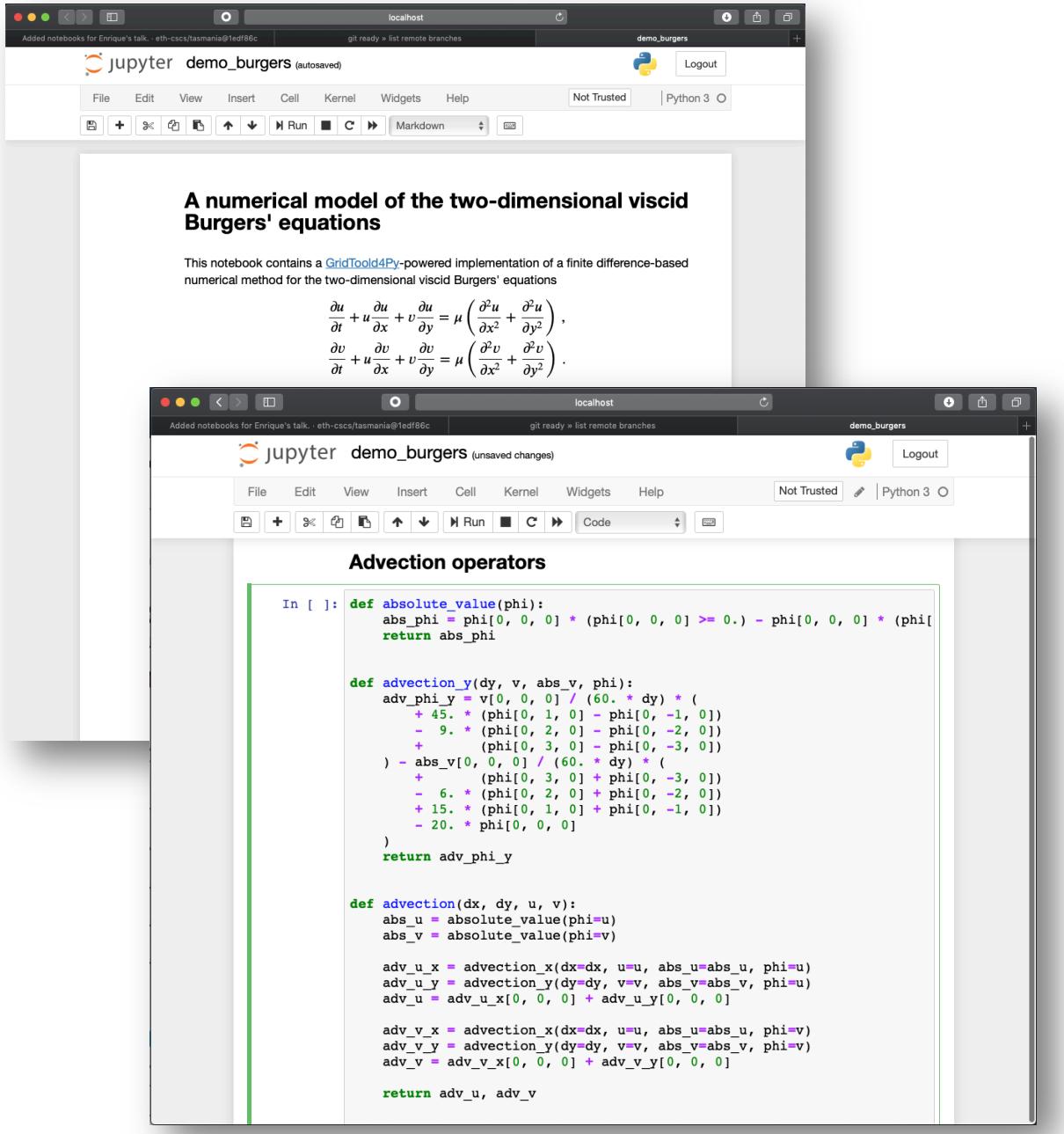
Python Ubiquity

Growing community in the atmospheric sciences

Lingua franca of atmospheric science and data science students

But… Can we achieve the 3 Ps?

- Performance – Run efficiently across range of systems (laptops to HPC systems)
- Portability – Map onto multiple hardware targets (CPUs, GPUs, …)
- Productivity – User code is a high-level specification of algorithm using Python



Python GridTools DSL

Python itself is too slow for execution on a supercomputer, so we use a DSL that generates GridTools C++ code

github.com/GridTools/gt4py

Active collaboration with ETH Zurich, CSCS, and MeteoSwiss on the further development of the package

Features

1. No explicit parallelism or optimization
2. Can execute in plain Python for testing / debugging
3. Writes to interface that compiler toolchain consumes and generates efficient low-level code

```python
import gt4py.gtscript as gtscript

@gtscript.function
def advection_x(dx, u, phi):
    adv_phi_x = u[0, 0, 0] / (60. * dx) * (
        + 45. * (phi[1, 0, 0] - phi[-1, 0, 0])
        -  9. * (phi[2, 0, 0] - phi[-2, 0, 0])
        +       (phi[3, 0, 0] - phi[-3, 0, 0])
    ) - abs(u[0, 0, 0]) / (60. * dx) * (
        +       (phi[3, 0, 0] + phi[-3, 0, 0])
        -  6. * (phi[2, 0, 0] + phi[-2, 0, 0])
        + 15. * (phi[1, 0, 0] + phi[-1, 0, 0])
        - 20. *  phi[0, 0, 0])
    return adv_phi_x

@gtscript.stencil(backend="numpy")
def advection(
    in_u: gtscript.Field[np.float64],
    out_adv: gtscript.Field[np.float64],
    *,
    dx: float
):
    with computation(PARALLEL), interval(...):
        out_adv = advection_x(dx=dx, u=in_u, phi=in_u)
```

Stencil Lifecycle

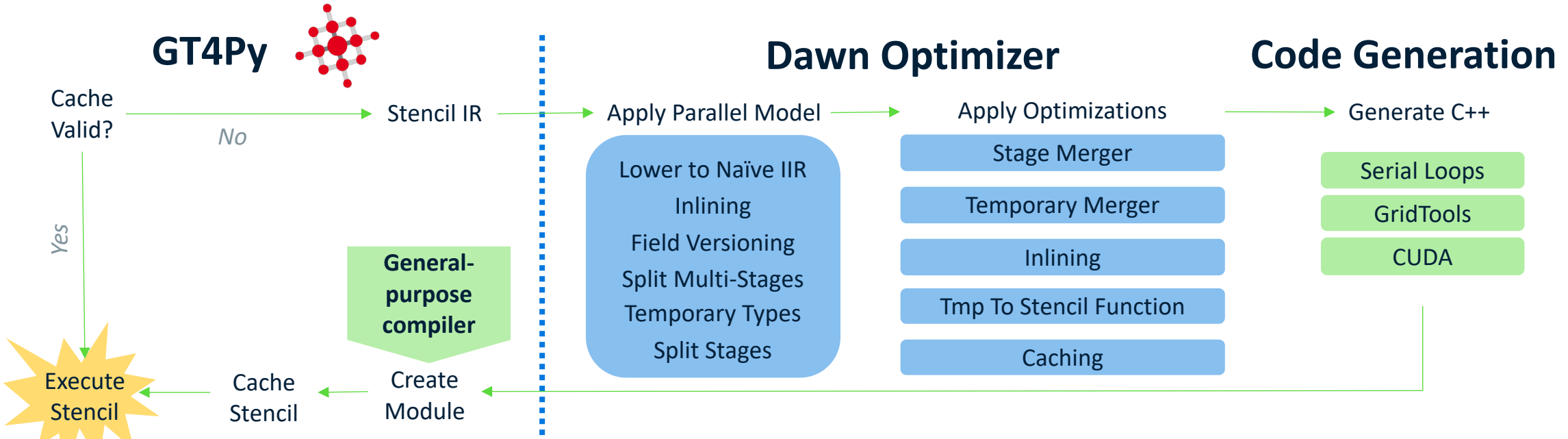
```
@gtscript.stencil(backend="dawn:cuda")
def advection(
    in_u: gtscript.Field[np.float64],
    out_adv: gtscript.Field[np.float64],
    dx: float):
    with computation(PARALLEL), interval(...):
        out_adv = advection_x(…)
```

GT4Py stencil pipeline initiated with python function decorator

Stencil Internal Representation (IR) - draft implementation of HIR standard

Calls Dawn with stencil IR as a JSON object through python interface

Generated code is inserted into pybind11 bindings to create loadable python module



Why DSL Compiler?

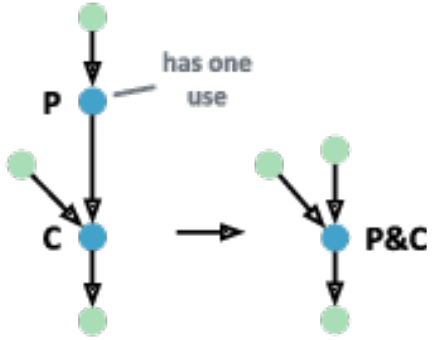
High-level compilers bridge the gap between readable and performant code

Separation of concerns between numerical model development and HPC

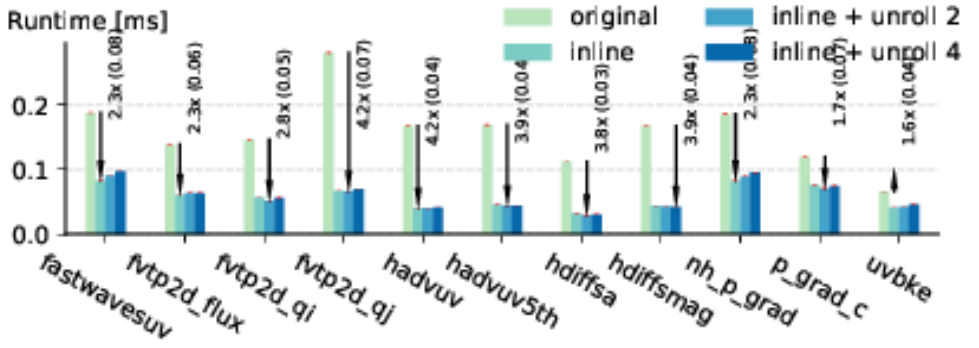
Can perform high level transformations that be difficult for general-purpose compiler, e.g.

- Merging stencils (+ inlining)
- Code reordering
- Field caching

Aggressive optimization: stencil fusion



Effect of inlining and unrolling on FV3 and COSMO stencil performance

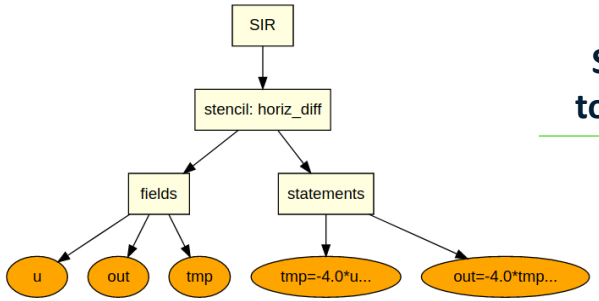


Domain-Specific Multi-Level IR Rewriting for GPU, Gysi, et. al. submitted

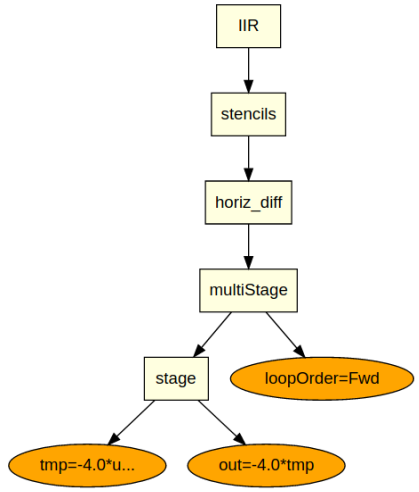
Example: Laplacian Stencil

```
@gtscript.stencil(backend="dawn:cuda")
def horiz_diff(u: Field, out: Field):
  with computation(PARALLEL), interval(…):
    tmp = -4.0 * u[0,0,0] + u[1,0,0] +
      u[-1,0,0] + u[0,1,0] + u[0,-1,0]
    out = -4.0 * tmp[0,0,0] + tmp[1,0,0] +
      u[-1,0,0] + tmp[0,1,0] + tmp[0,-1,0]
```

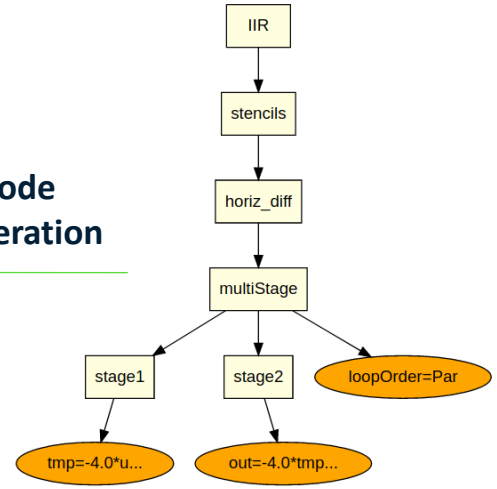
GT4Py to SIR



SIR to IIR



Stage Splitting



Code Generation

Temporary Caching

```
__global__ void horiz_diff(const double* u, double* out) {
  __shared__ double tmpCache[...];
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const int j = blockIdx.y * blockDim.y + threadIdx.y;
  for(int k = 0; k <= kMax; ++k) {
    if (i < iMax+1 && j < jMax+1) {
      tmp_cache[i,j] = -4.0 * u[i,j,k] + u[i+1,j,k] +
        u[i-1,j,k] + u[i,j+1,k] + u[i,j-1,k];
    }
  __syncthreads();
  if (i < iMax && j < jMax) {
    out[i,j,k] = -4.0 * tmpCache[i,j] + tmpCache[i+1,j] +
      tmpCache[i-1,j] + tmpCache[i,j+1] + tmpCache[i,j-1];
    }
  }
}
```

Target Compiler

FV3 Optimization

```
def update_dz_c(dp_ref: Field, zs: Field, area: Field,
    ut: Field, vt: Field, gz: Field, gz_x: Field,
    gz_y: Field, ws3: Field, *, dt: float):
    with computation(PARALLEL):
        with interval(0, 1):
            xfx = p_weighted_average_top(ut, dp_ref)
            yfx = p_weighted_average_top(vt, dp_ref)
        with interval(-1, None):
            xfx = p_weighted_average_bottom(ut, dp_ref)
            yfx = p_weighted_average_bottom(vt, dp_ref)
        with interval(1, -1):
            xfx = p_weighted_average_domain(ut, dp_ref)
            yfx = p_weighted_average_domain(vt, dp_ref)
    with computation(PARALLEL), interval(...):
        fx, fy = xy_flux(gz_x, gz_y, xfx, yfx)
        gz = (gz_y * area + flux_divergence(fx, fy)) /
             (area + flux_divergence(xfx, xfy))
    with computation(PARALLEL), interval(-1, None):
        rdt = 1.0 / dt
        ws3 = (zs - gz) * rdt
    with computation(BACKWARD), interval(0, -1):
        gz_kp1 = gz[0, 0, 1] + DZ_MIN
        gz = gz if gz > gz_kp1 else gz_kp1
```

Initial implementation of update on C-grid (right) would result in four “multistages” or repeated GPU kernel calls

Using aggressive optimization in Dawn, this results in a single backward (in k) multistage with 7 stages

```
Stage 0 Do (end-1:end)
    xfx = p_weighted_average_bottom(ut, dp_ref)
    yfx = p_weighted_average_bottom(vt, dp_ref)
Stage 0 Do (start:start+1)
    xfx = p_weighted_average_top(ut, dp_ref)
    yfx = p_weighted_average_top(vt, dp_ref)
Stage 1 Do (start+1:end-1)
    xfx = p_weighted_average_domain(ut, dp_ref)
    yfx = p_weighted_average_domain(vt, dp_ref)
Stage 2 Do (start:end)
    fx, fy = xy_flux(gz_x, gz_y, xfx, yfx)
Stage 3 Do (start:end)
    gz_0 = (gz_y * area + flux_divergence(fx, fy)) /
      (area + flux_divergence(xfx, xfy))
Stage 4 Do (end-1:end)
    gz_0 = gz
Stage 5 Do (start:end-1)
    gz_kp1 = gz[0, 0, 1] + DZ_MIN
    gz = gz if gz > gz_kp1 else gz_kp1
Stage 6 Do (end-1:end)
    ws3 = (zs - gz_0) * 1.0 / dt
```

Porting Infrastructure

Developed test/validation framework using Serialbox

github.com/GridTools/serialbox

Versioned serialization data is kept in cloud buckets

Pytest-driven framework while developing stencils proven invaluable for catching errors

dyn_core.F90

```
...
!$ser savepoint PGradC-In
!$ser data delpc=delpc pkc=pkc gz=gz uc=uc vc=vc
call p_grad_c(delpc, pkc, gz, uc, vc, ...)
!$ser savepoint PGradC-Out
!$ser data uc=uc vc=vc
...
```

test_pgradc.py

```
import numpy as np
import serializer

...

# read serialized inputs
sp = 'PGradC-In'
delpc = serializer.read('delpc', sp)
pkc   = serializer.read('pkc', sp)
gz    = serializer.read('gz', sp)
uc    = serializer.read('uc', sp)
vc    = serializer.read('vc', sp)

# run computation
uc, vc = p_grad_c(delpc, pkc, gz, uc, vc, ...)

# read serialized outputs
sp = 'PGradC-Out'
uc_ref = serializer.read('uc', sp)
vc_ref = serializer.read('vc', sp)

# check result
eps = 3.0e-6
assert np.all(np.abs(uc - uc_ref) < eps)
assert np.all(np.abs(vc - vc_ref) < eps)
```

Progress So Far

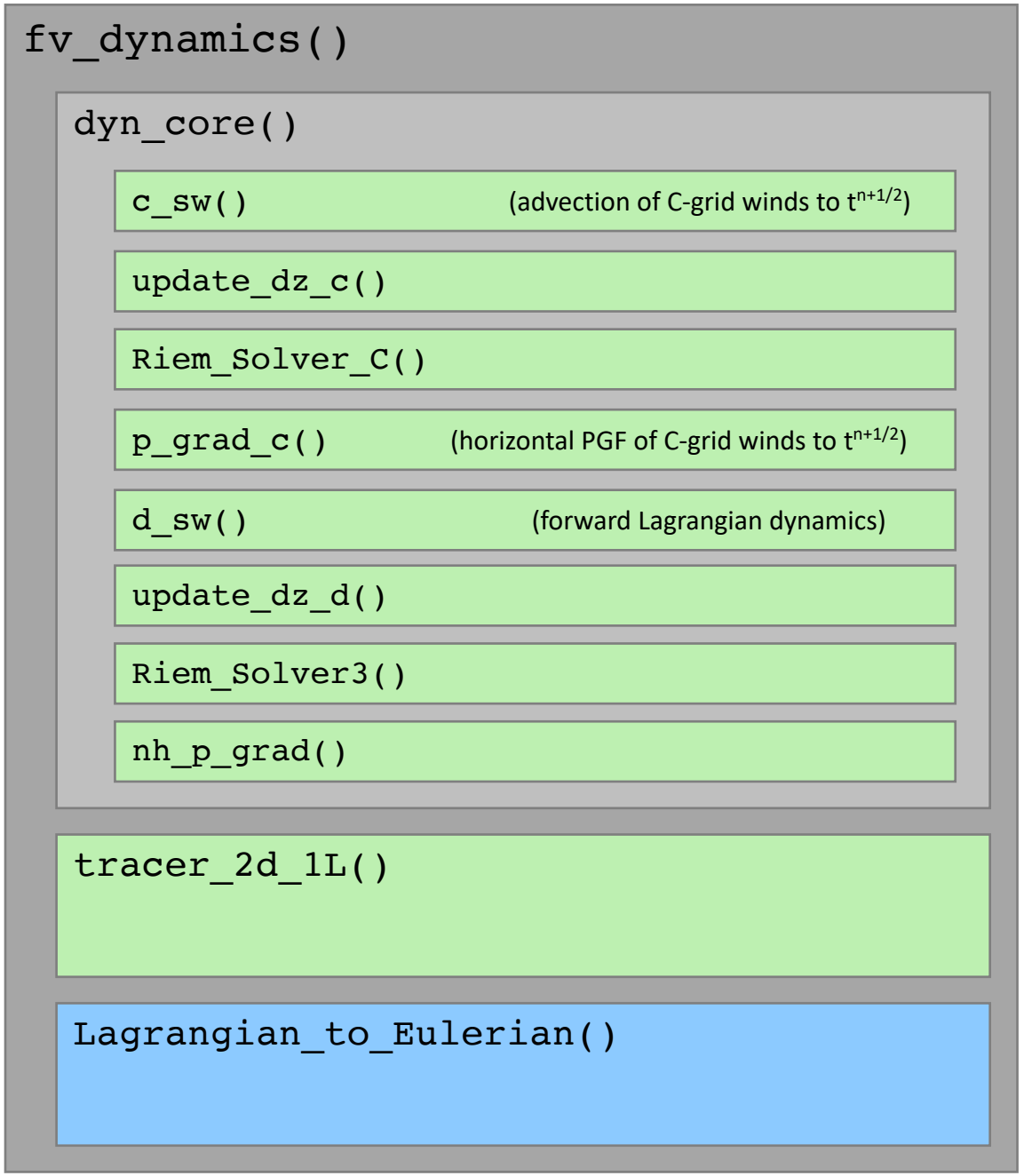
Focus on non-hydrostatic configuration

Finishing up port of tracer stencil, work now focuses on linking these together

Certain motifs are not well covered yet by GT4Py, including:

- Corners and edges
- Loops (Lagrangian remapping)
- Directional offsets
- Neumann boundary conditions

We are finding work-arounds and/or implementing in the DSL



Motif: Corners and Edges

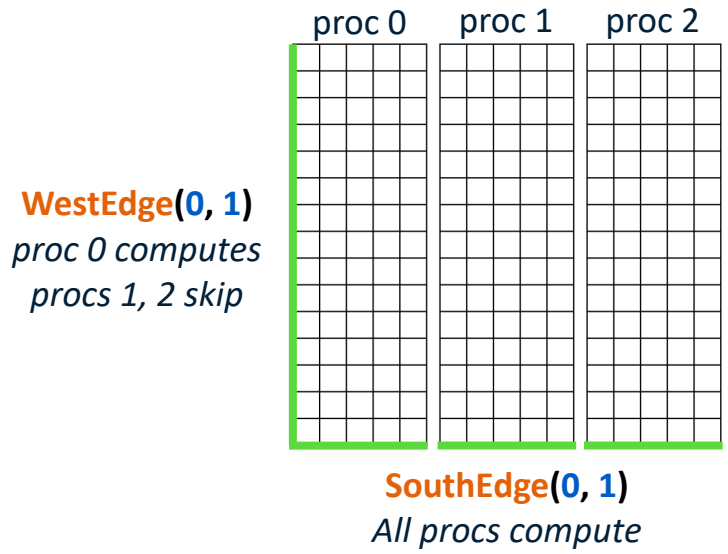
GT4Py will soon have a method of defining computation on specific sub-regions of the horizontal iteration space

Concept covers boundary conditions even in the event of a distributed domain

Callbacks to application code generate iteration ranges at stencil compile time – do not show up in the generated code

- Domain decomp is transparent to GT4Py and backend
- Region objects have common interface that GT4Py uses

```
@gtscript.stencil(backend=backend_type)
def ubke(uc: Field, vc: Field, cosa: Field, rsina: Field,
    ut: Field, ub: Field, dt5: float, dt4: float
):
    with computation(PARALLEL), interval(...):
        ub = dt5 * (uc[0, -1, 0] + uc -
            (vc[-1, 0, 0] + vc) * cosa) * rsina
    with computation(PARALLEL), region(WestEdge(0, 1),
        EastEdge(0, 1)), interval(...):
        ub = dt5 * (ut[0, -1, 0] + ut)
    with computation(PARALLEL), region(SouthEdge(0, 1),
        NorthEdge(0, 1)), interval(...):
        ub = dt4 * (-ut[0, -2, 0] + 3.0 *
            (ut[0, -1, 0] + ut) - ut[0, 1, 0])
```



Summary and Ongoing Work

Effective collaboration with partner organizations GFDL, MeteoSwiss, ETH Zurich

FV3

- Nearly finished initial port of FV3 dynamical core in GT4Py, validated against Fortran reference
- Enlarging stencils to expose more optimization potential to the DSL toolchain

Toolchain

- Significantly refactored Dawn, is now connected with GT4Py for a seamless development environment
- Analyzing performance of stencils and adding optimizations for new GPU hardware
- Expanding DSL language to support missing motifs
- Improving the debugging experience for users

VCM DSL Group

Oliver Elbert (GFDL)

Oli Fuhrer

Johann Dahm

Tobias Wicky

Rhea George

Eddie Davis

Jeremy McGibbon

Mark Cheeseman