

Compiler architecture: the discipline and abstractions

to exploit application domains

to target accelerators

to feed microarchitecture

Paul Kelly

Group Leader, Software Performance Optimisation, Department of Computing
Imperial College London

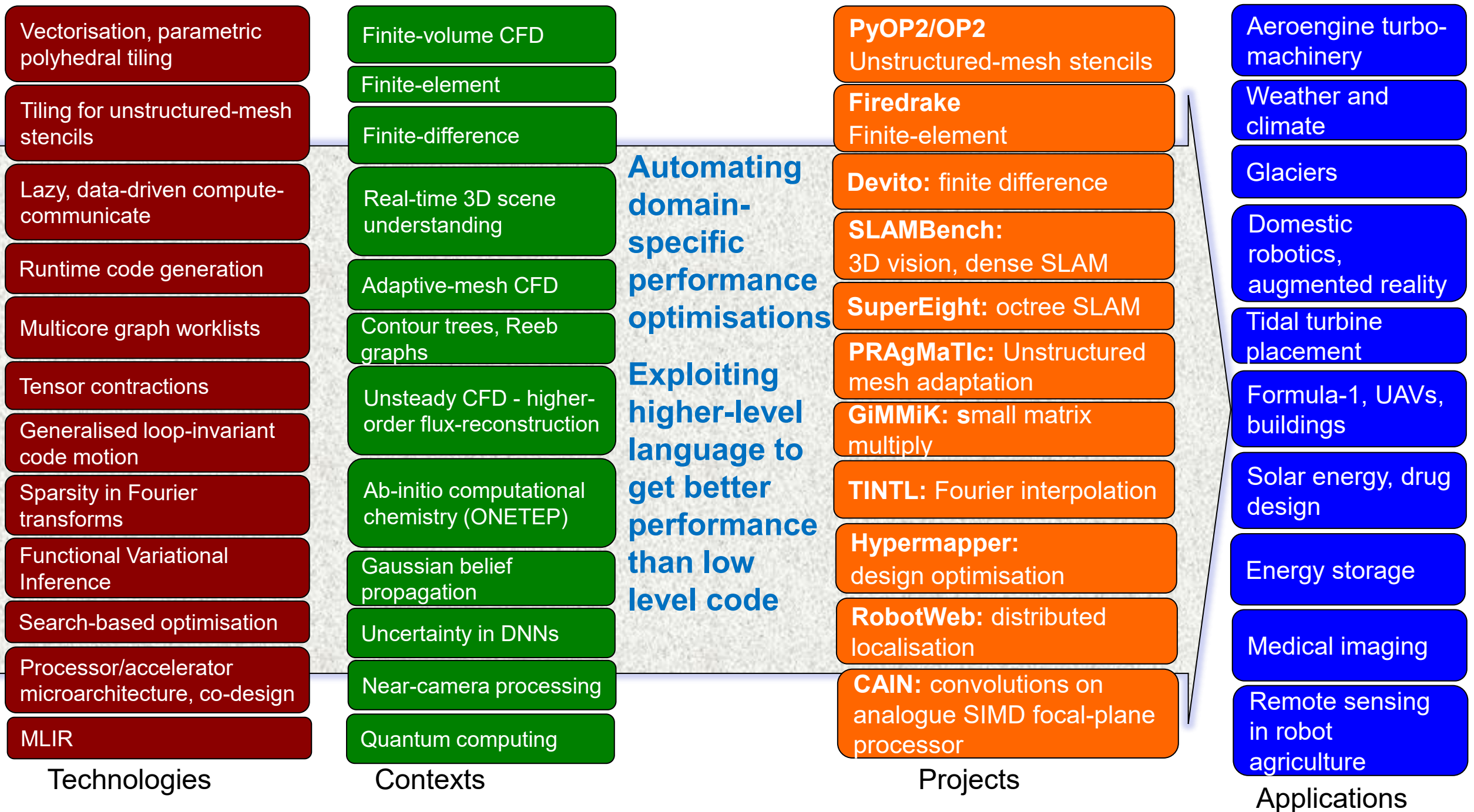
Joint work with Luke Panayi, Martin Berger, Rohan Gandhi, Jim Whittaker, Vassilios Chouliaras, Andrei Sburlan, Avaneesh Deleep, George Bisbas, Edward Stow, Jacky Wong, David Ham, Fabio Luporini, Lawrence Mitchell, Graham Markall, Mike Giles, Gerard Gorman, Florian Rathgeber, Luigi Nardi, Carlos Cueto, Lluís Guasch, Fabio Luporini, Oscar Bates, George Strong, Oscar Calderon Agudo, Javier Cudeiro, Gerard Gorman, and Meng-Xing Tang, Hidenobu Matsuki, Riku Murai, Andrew J. Davison, David Pearce, Piotr Dudek, Tobias Grosser and many more

Who am I and what do I do?

- I've worked on a lot of things....
 - GPUs, FPGAs, cache coherency, vectorisation, verification, bounds checking, CPU architecture...
- I have worked on general-purpose compilers
 - Notably pointer analysis
 - adopted into GCC
 - (actually the work of my PhD graduate David Pearce)
- But the benefits were incremental



- Meanwhile I engaged with applications specialists
 - Who know they have major performance optimisation opportunities
 - So I got interested in automating domain-specific optimisations
- Which grew into deep engagements with projects like
 - **Firedrake**, **Devito** and **PyFR**, that automate the pathway from PDE to high-performance code for laptops and supercomputers
 - Robot vision, SLAM, robot localisation



What are we doing *next* ?

- Gaussian Belief Propagation as a foundation for managing locality, distribution, asynchrony and approximation in spatial AI
- Gaussian Splatting SLAM: capturing manipulable photorealistic scenes in real time
- Quantum circuits as a DSL
- XDSL and MLIR: common compiler architecture ecosystem for DSLs
- Instruction labelling: policy/prediction hints for scalable microarchitecture
- Tensor contractions as a compiler IR
- On-sensor vision computing/accelerator architectures and programming models
- Compiler technology for simulating quantum computers

This talk

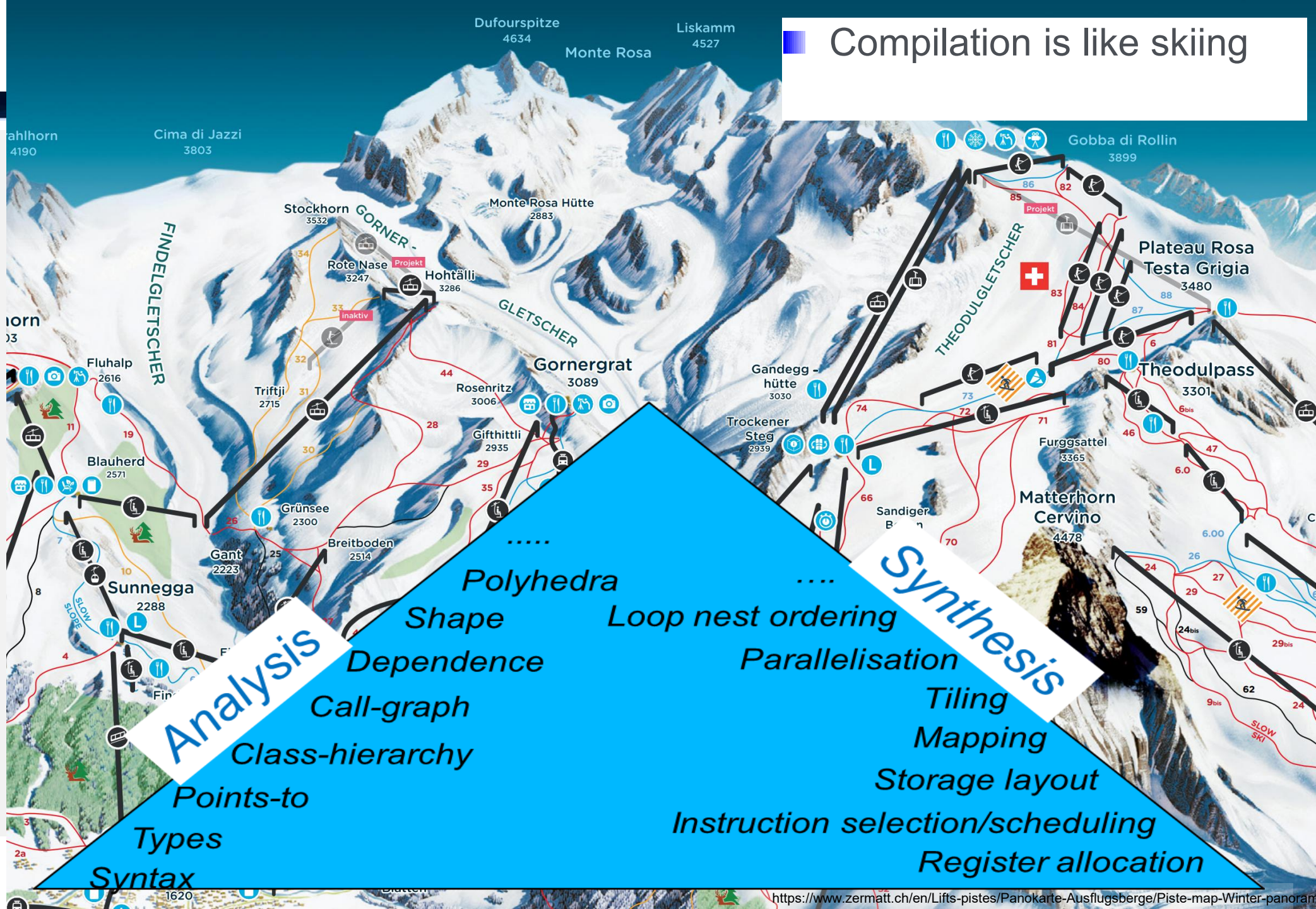
- **Compiler *architecture***
- **The importance of capturing application code at the highest-possible level of representation**
- **Getting the abstraction right**
 - **To capture what the code is trying to do**
 - **To capture what we need to optimise for the hardware**
 - **For CPUs**
 - **For GPUs**
 - **For custom accelerators**

- Alan Turing realised we could use digital technology to implement any computable function
- He then proposed the idea of a “universal” computing device – a *single* device which, with the right program, can implement any computable function *without further configuration*
- “**Turing Tax**”, or “**Turing Tariffs**”: the overhead (performance, cost, or energy) of universality in this sense
- The performance (time/area/energy) difference between a **special-purpose** device and a **general-purpose** one
- One of the fundamental questions of computer architecture is to how to reduce the Turing Tax

■ Compilation is like skiing

■ What about
compilers
?

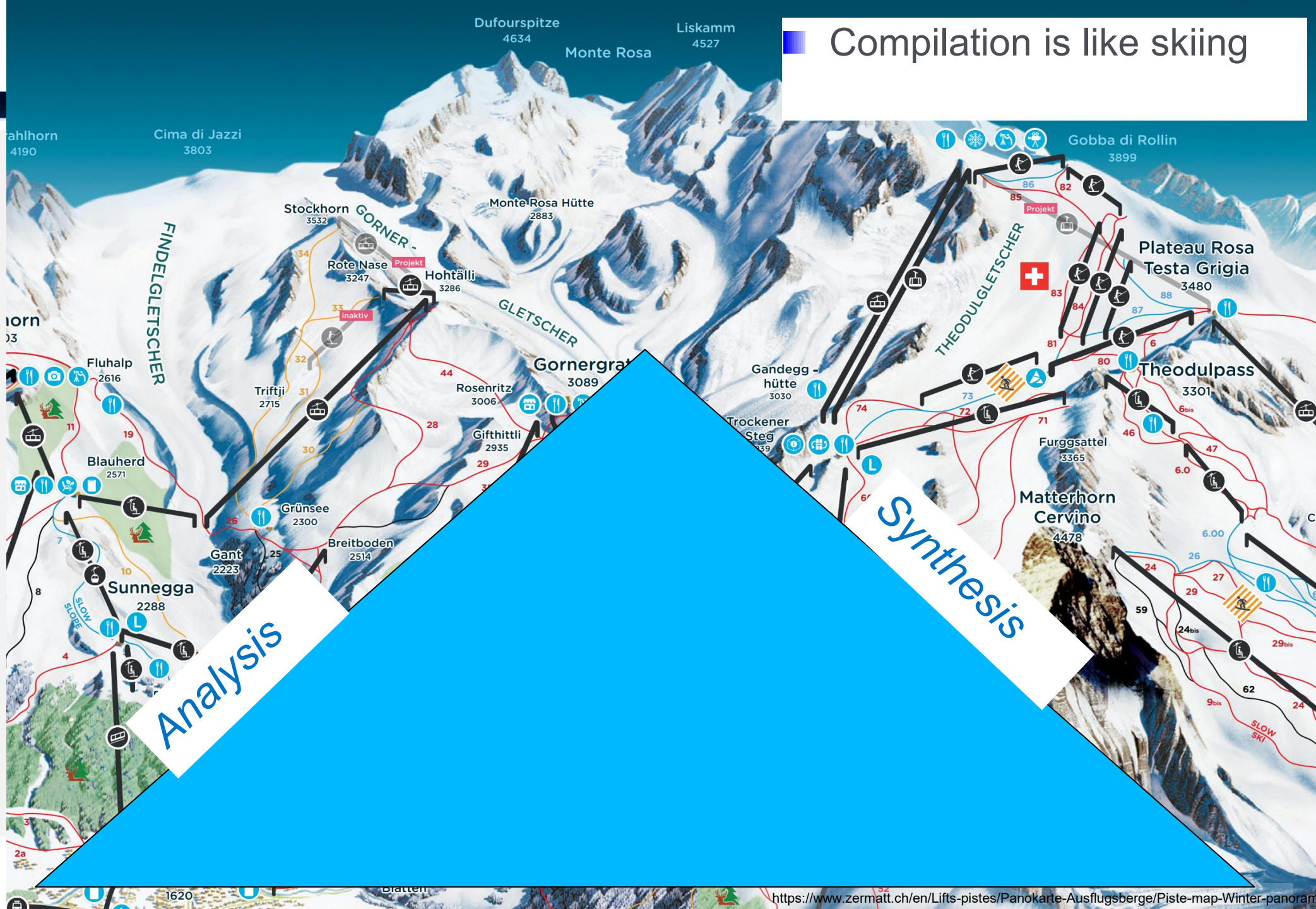
■ Is there a
Turing Tax
for
compilers
too?



What about
compilers
?

Is there a
Turing Tax
for
compilers
too?

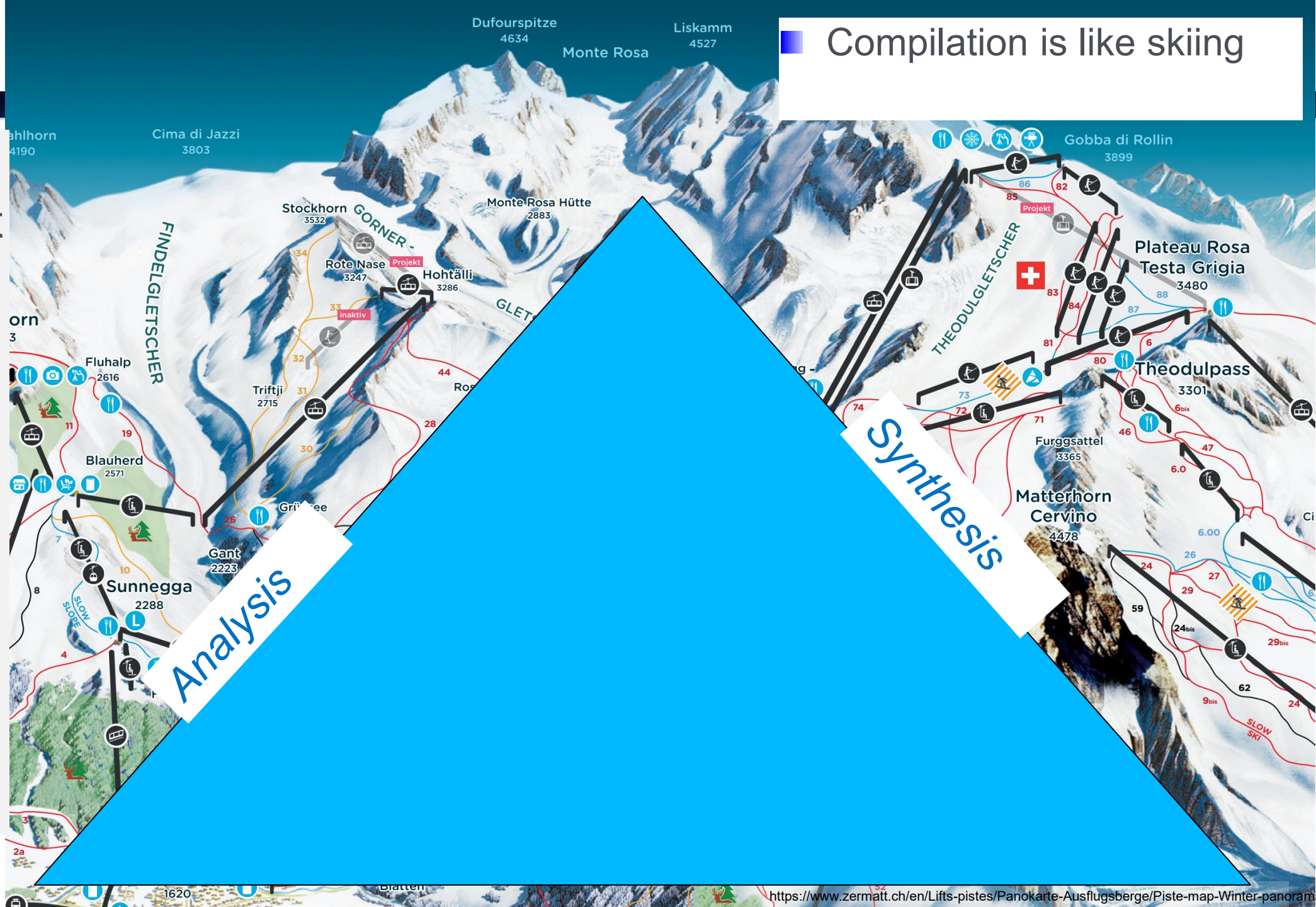
Compilation is like skiing



■ Compilation is like skiing

■ What about
compilers
?

■ Is there a
Turing Tax
for
compilers
too?





■ Compilation is like skiing

■ What about
compilers?

■ Is there a
Turing Tax
for
compilers
too?

Carrying your skis up the
mountain isn't the best bit

Synthesis

What about
compilers?

The price you
pay for
coding in a
general-
purpose
language

When you
could have
used a DSL



Compilation is like skiing

Carrying your skis up the
mountain isn't the best bit
All of this is Turing Tariff!

Synthesis



Firedrake

[Documentation](#) [Install](#) [Team](#) [Citing](#) [Publications](#) [Events](#) [Funding](#) [Contact](#) [GitHub](#)

Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM). Firedrake uses sophisticated code generation to provide mathematicians, scientists, and engineers with a very high productivity way to create sophisticated high performance simulations.

Features:

- Expressive specification of any PDE using the Unified Form Language from the **FEniCS Project**.
- Sophisticated, programmable solvers through seamless coupling with **PETSc**.
- Triangular, quadrilateral, and tetrahedral unstructured meshes.
- Layered meshes of triangular wedges or hexahedra.
- Vast range of finite element spaces.
- Sophisticated automatic optimisation, including sum factorisation for high order elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.

Example DSL #1

Latest commits to the Firedrake main branch on Github

Merge pull request #4536 from firedrakeproject/JHopeCollins/merge-release-into-main

Connor Ward authored at 05/09/2025, 15:47:46

Merge branch 'main' into JHopeCollins/merge-release-into-main

Josh Hope-Collins authored at 05/09/2025, 14:34:12

Fix docs build (#4537)

Connor Ward authored at 05/09/2025, 14:31:29

MG: rediscretize with a different coarse_mat_type (#4538)

Pablo Brubeck authored at 05/09/2025, 10:35:14

Remove `Interpolator.interpolate` (#4531)

Leo Collins authored at 04/09/2025, 16:34:07

Active team members



David A. Ham



Paul H. J. Kelly



Colin J. Cotter



Robert C. Kirby



Koki Sagiyama



Jack Betteridge



Daniel R. Shapero



Connor J. Ward



Patrick E. Farrell



Pablo D. Brubeck



India Marsden



Daisuke I. Doli



Joshua Hope-Collins



Umberto Zerbini

Former team members



Lawrence Mitchell



Reuben W. Nixon-Hill



Nacime Bouziani



Sophia Vordenwuebbe



Thomas H. Gibson



Miklós Homolya



Tianjiao Sun



Andrew T. T. McRae



Fabio Luporini



Alastair Gregory



Michael Lange



Simon W. Funke



Florian Rathgeber



Ogeorge-Teodor Bercea



Graham R. Markall

■ Many projects build on Firedrake, for example:

- **Firedrake-adjoint**: extension of the pyadjoint algorithmic differentiation framework to yield fully automated derivation of adjoint PDE solvers
- **Irksome**: Automates Runge–Kutta time-stepping
- **AsQ**: parallel-in-time
- **Slate**: linear algebra on finite element tensors, for hybridisation, static condensation
- **PCPATCH**: topological construction of multigrid relaxation methods

- **Defcon**: deflated continuation method for computing bifurcation diagrams
- Goal-oriented mesh adaptation
- Integration with PyTorch and JAX

Including tools for specific application domains, for example:

- **Thetis**: unstructured grid coastal modelling framework
- **IcePack**: glacier flow
- **Gusto**: dynamical cores for weather prediction and climate models

Firedrake example: Burgers equation

$$\int_{\Omega} \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla) u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \, dx = 0.$$

- From the weak form of the PDE, we derive an equation to solve, that determines the state at each timestep in terms of the previous timestep

- Transcribe into Python: u is u^{n+1} , $u_$ is u^n :

```
F = (inner((u - u_)/timestep, v)
      + inner(dot(u, nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
```

- Set up the equation and solve for the next timestep u :

```
solve(F == 0, u)
```

- At this point, Firedrake generates code to assemble a linear system, runs it and calls a linear solver (we use PetSC)

This is the
helicopter ride



Burgers equation

```
from firedrake import *
n = 50
mesh = UnitSquareMesh(n, n)

# We choose degree 2 continuous Lagrange polynomials. We also need a
# piecewise linear space for output purposes::

V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)

# We also need solution functions for the current and the next timestep::

u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")

v = TestFunction(V)

# We supply an initial condition::

x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)

# Start with current value of u set to the initial condition, and use the
# initial condition as our starting guess for the next value of u::

u_.assign(ic)
u.assign(ic)

# :math:`\nu` is set to a (fairly arbitrary) small constant value::

nu = 0.0001

timestep = 1.0/n

# Define the residual of the equation::

F = (inner((u - u_)/timestep, v)
      + inner(dot(u, nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx

outfile = File("burgers.pvd")

outfile.write(project(u, V_out, name="Velocity"))

# Finally, we loop over the timesteps solving the equation each time::

t = 0.0
end = 0.5
while (t <= end):
    solve(F == 0, u)
    u_.assign(u)
    t += timestep
    outfile.write(project(u, V_out, name="Velocity"))
```

- Complete runnable Python script
 - sets up the equation,
 - specifies discretisation
 - Iterates multiple timesteps
 - Writes velocity field out for animated visualisation
- Firedrake implements the Unified Form Language (UFL)
- Embedded in Python

This is the
helicopter ride



- (UFL is also the DSL of the FEniCS project)

```

from firedrake import *
n = 50
mesh = UnitSquareMesh(n, n)

# We choose degree 2 continuous Lagrange polynomials. We also need a
# piecewise linear space for output purposes::

V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)

# We also need solution functions for the current and the next timestep::

u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")

v = TestFunction(V)

# We supply an initial condition::

x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)

# Start with current value of u set to the initial condition, and use the
# initial condition as our starting guess for the next value of u::

u_.assign(ic)
u.assign(ic)

# :math:`\nu` is set to a (fairly arbitrary) small constant value::

nu = 0.0001

timestep = 1.0/n

# Define the residual of the equation::

F = (inner((u - u_)/timestep, v)
      + inner(dot(u, nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx

outfile = File("burgers.pvd")

outfile.write(project(u, V_out, name="Velocity"))

# Finally, we loop over the timesteps solving the equation each time::

t = 0.0
end = 0.5
while (t <= end):
    solve(F == 0, u)
    u_.assign(u)
    t += timestep
    outfile.write(project(u, V_out, name="Velocity"))

```

```
mesh = UnitSquareMesh(n, n)
```

```
V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)
```

```
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")
```

set up initial conditions for u and u_

Define the residual of the equation::

```
F = (inner((u - u_)/timestep, v)
      + inner(dot(u, nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
```

```

t = 0.0
end = 0.5
while (t <= end):
    solve(F == 0, u)
    u_.assign(u)
    t += timestep
    outfile.write(project(u, V_out, name="Velocity"))

```

Time-stepping loop:

- Solve for the state at the next timestep
- Save snapshot to file for animation

```
#include <math.h>
#include <petsc.h>

void wrap_form00_cell_integral_otherwise(int const start, int const end, Mat const mat0, double const * __restrict__ dat1, double const * __restrict__ dat0, int const * __restrict__ map0, int const * __restrict__ map1)
{
    double form_t0...t16;
    double const form_t17[7] = { ... };
    double const form_t18[7 * 6] = { ... };
    double const form_t19[7 * 6] = { ... };
    double form_t2;
    double const form_t20[7 * 6] = { ... };
    double form_t21...t37;
    double form_t38[6];
    double form_t39[6];
    double form_t4;
    double form_t40...t45;
    double form_t5...t9;
    double t0[6 * 2];
    double t1[3 * 2];
    double t2[6 * 2 * 6 * 2];

    for (int n = start; n <= -1 + end; ++n)
    {
        for (int i4 = 0; i4 <= 5; ++i4)
            for (int i5 = 0; i5 <= 1; ++i5)
                for (int i6 = 0; i6 <= 5; ++i6)
                    for (int i7 = 0; i7 <= 1; ++i7)
                        t2[24 * i4 + 12 * i5 + 2 * i6 + i7] = 0.0;
        for (int i2 = 0; i2 <= 2; ++i2)
            for (int i3 = 0; i3 <= 1; ++i3)
                t1[2 * i2 + i3] = dat1[2 * map1[3 * n + i2] + i3];
        for (int i0 = 0; i0 <= 5; ++i0)
            for (int i1 = 0; i1 <= 1; ++i1)
                t0[2 * i0 + i1] = dat0[2 * map0[6 * n + i0] + i1];
        form_t0 = -1.0 * t1[1];
        form_t1 = form_t0 + t1[3];
        form_t2 = -1.0 * t1[0];
        form_t3 = form_t2 + t1[2];
        form_t4 = form_t0 + t1[5];
        form_t5 = form_t2 + t1[4];
        form_t6 = form_t3 * form_t4 + -1.0 * form_t5 * form_t1;
        form_t7 = 1.0 / form_t6;
        form_t8 = form_t7 * -1.0 * form_t1;
        form_t9 = form_t4 * form_t7;
        form_t10 = form_t3 * form_t7;
        form_t11 = form_t7 * -1.0 * form_t5;
        form_t12 = 0.0001 * (form_t8 * form_t9 + form_t10 * form_t11);
        form_t13 = 0.0001 * (form_t8 * form_t8 + form_t10 * form_t10);
        form_t14 = 0.0001 * (form_t9 * form_t9 + form_t11 * form_t11);
        form_t15 = 0.0001 * (form_t9 * form_t8 + form_t11 * form_t10);
        form_t16 = fabs(form_t6);
        for (int form_ip = 0; form_ip <= 6; ++form_ip)
        {
            form_t26 = 0.0; form_t25 = 0.0; form_t24 = 0.0; form_t23 = 0.0; form_t22 = 0.0; form_t21 = 0.0;
            for (int form_i = 0; form_i <= 5; ++form_i)
            {
                form_t21 = form_t21 + form_t20[6 * form_ip + form_i] * t0[1 + 2 * form_i];
                form_t22 = form_t22 + form_t19[6 * form_ip + form_i] * t0[1 + 2 * form_i];
                form_t23 = form_t23 + form_t20[6 * form_ip + form_i] * t0[2 * form_i];
                form_t24 = form_t24 + form_t19[6 * form_ip + form_i] * t0[2 * form_i];
                form_t25 = form_t25 + form_t18[6 * form_ip + form_i] * t0[1 + 2 * form_i];
                form_t26 = form_t26 + form_t18[6 * form_ip + form_i] * t0[2 * form_i];
            }
            form_t27 = form_t17[form_ip] * form_t16;
            form_t28 = form_t27 * form_t15;
            form_t29 = form_t27 * form_t14;
            form_t30 = form_t27 * (form_t26 * form_t9 + form_t25 * form_t11);
            form_t31 = form_t27 * form_t13;
            form_t32 = form_t27 * form_t12;
            form_t33 = form_t27 * (form_t26 * form_t8 + form_t25 * form_t10);
            form_t34 = form_t27 * (form_t11 * form_t24 + form_t10 * form_t23);
            form_t35 = form_t27 * (form_t9 * form_t22 + form_t8 * form_t21);
            form_t36 = form_t27 * (50.0 + form_t9 * form_t24 + form_t8 * form_t23);
            form_t37 = form_t27 * (50.0 + form_t11 * form_t22 + form_t10 * form_t21);
            for (int form_k0 = 0; form_k0 <= 5; ++form_k0)
            {
                form_t38[form_k0] = form_t18[6 * form_ip + form_k0] * form_t37;
                form_t39[form_k0] = form_t18[6 * form_ip + form_k0] * form_t36;
            }
            for (int form_j0 = 0; form_j0 <= 5; ++form_j0)
            {
                form_t40 = form_t18[6 * form_ip + form_j0] * form_t35;
                form_t41 = form_t18[6 * form_ip + form_j0] * form_t34;
                form_t42 = form_t20[6 * form_ip + form_j0] * form_t31 + form_t18[6 * form_ip + form_j0] * form_t33 + form_t19[6 * form_ip + form_j0] * form_t32;
                form_t43 = form_t20[6 * form_ip + form_j0] * form_t28 + form_t18[6 * form_ip + form_j0] * form_t30 + form_t19[6 * form_ip + form_j0] * form_t29;
                for (int form_k0_0 = 0; form_k0_0 <= 5; ++form_k0_0)
                {
                    form_t44 = form_t43 * form_t19[6 * form_ip + form_k0_0];
                    form_t45 = form_t42 * form_t20[6 * form_ip + form_k0_0];
                    t2[24 * form_j0 + 2 * form_k0_0] = t2[24 * form_j0 + 2 * form_k0_0] + form_t45 + form_t18[6 * form_ip + form_j0] * form_t39[form_k0_0] + form_t44;
                    t2[13 + 24 * form_j0 + 2 * form_k0_0] = t2[13 + 24 * form_j0 + 2 * form_k0_0] + form_t30 + form_t18[6 * form_ip + form_j0] * form_t41;
                    t2[12 + 24 * form_j0 + 2 * form_k0_0] = t2[12 + 24 * form_j0 + 2 * form_k0_0] + form_t18[6 * form_ip + form_k0_0] * form_t40;
                }
            }
        }
        MatSetValuesBlockedLocal(mat0, 6, &(map0[6 * n]), 6, &(map0[6 * n]), &(t2[0]), ADD_VALUES);
    }
}
```

Gather data from
neighbouring cells using
adjacency graph

Compute re-used
common sub-terms

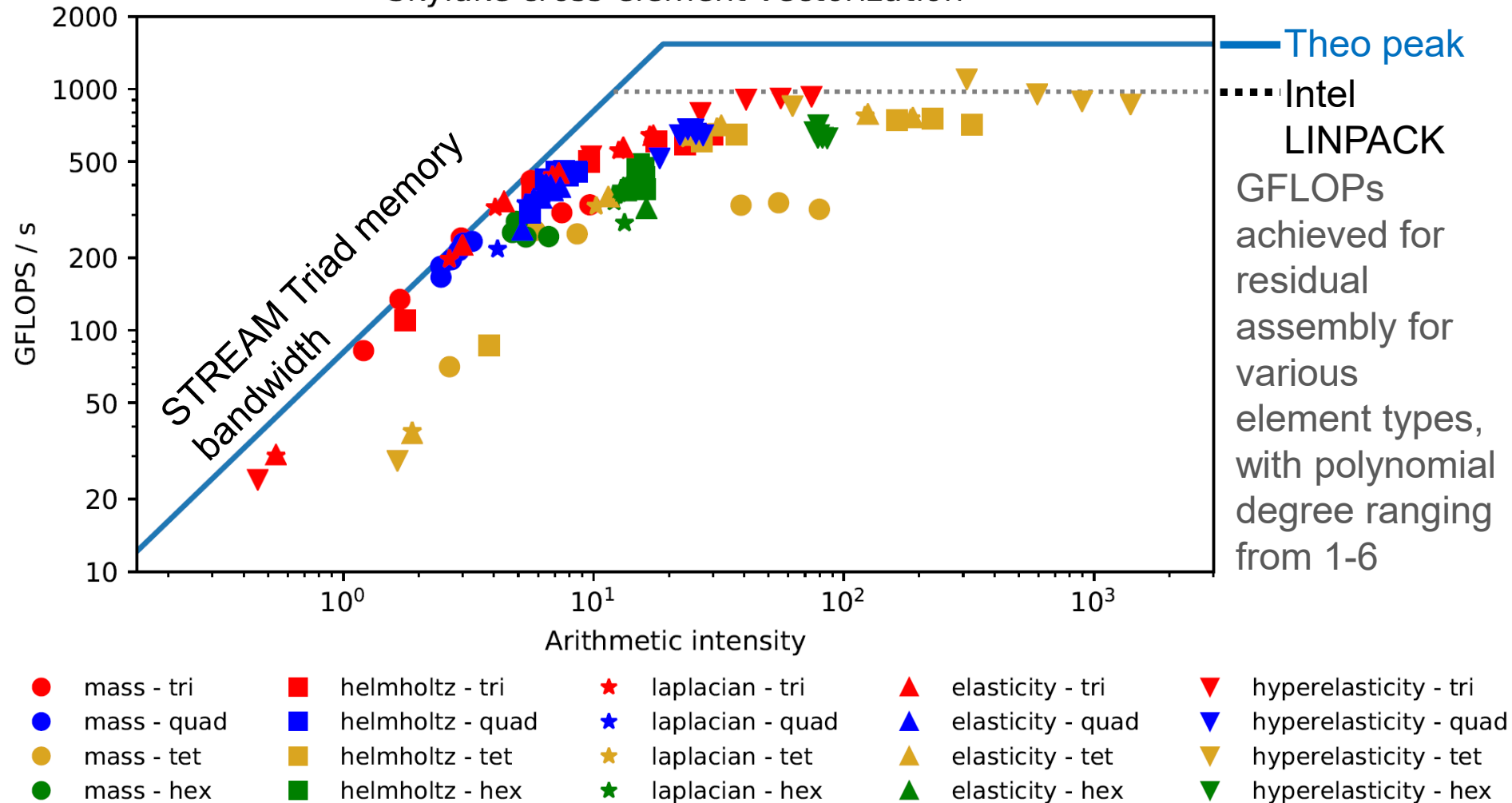
Compute integrals using
polynomial approximation
of fluid state in this cell

Sum local contributions
into global system matrix

- Generated code to assemble the resulting linear system matrix
- Executed at each triangle in the mesh
- Accesses degrees of freedom shared with neighbour triangles through indirection map

■ Does it generate good code?

Skylake cross-element vectorization



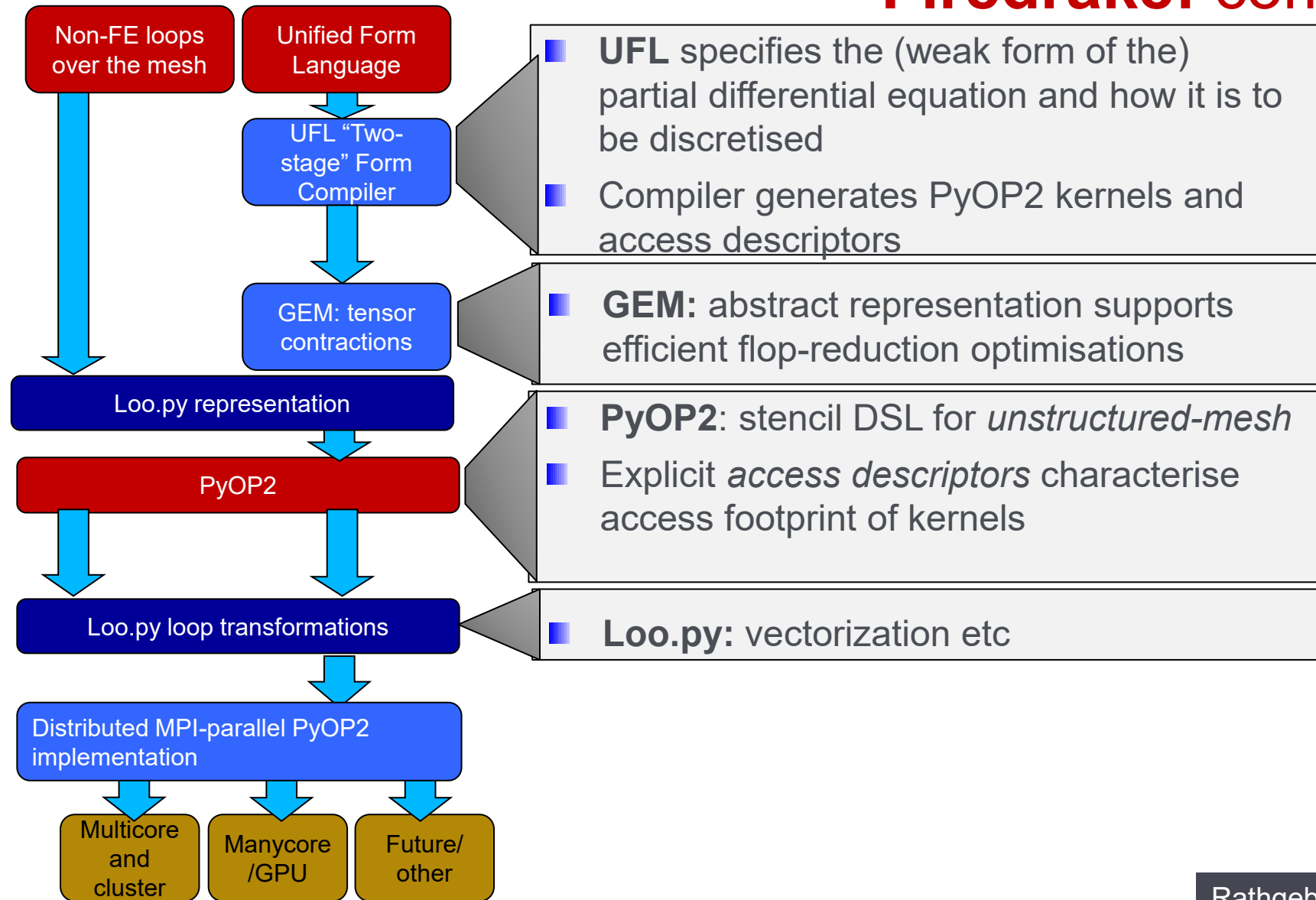
[Skylake Xeon Gold 6130 (on all 16 cores, 2.1GHz, turboboost off, Stream: 36.6GB/s, GCC7.3 -march=native)]

The big story here:

- Five different PDEs
- Six different choices for polynomial degree
- Automatically compiled
- So we see how performance varies across the benchmark suite

A study of vectorization for matrix-free finite element methods, Tianjiao Sun et al
IJHPCA 2020 <https://arxiv.org/abs/1903.08243>

Firedrake: compiler architecture



- **Sequence of intermediate representations**
- **100% Python**
- **Runtime code generation, code-caching**

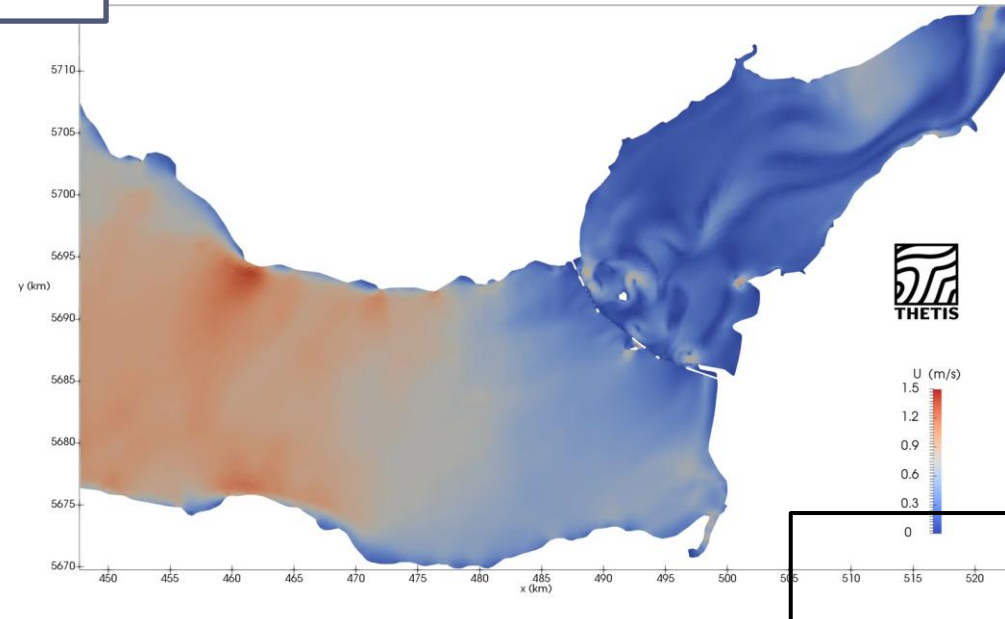
In production In advanced development Some prototyping

Rathgeber, Ham, Mitchell et al, ACM TOMS 2016,
Tianjiao Sun et al
<https://arxiv.org/pdf/1903.08243.pdf>

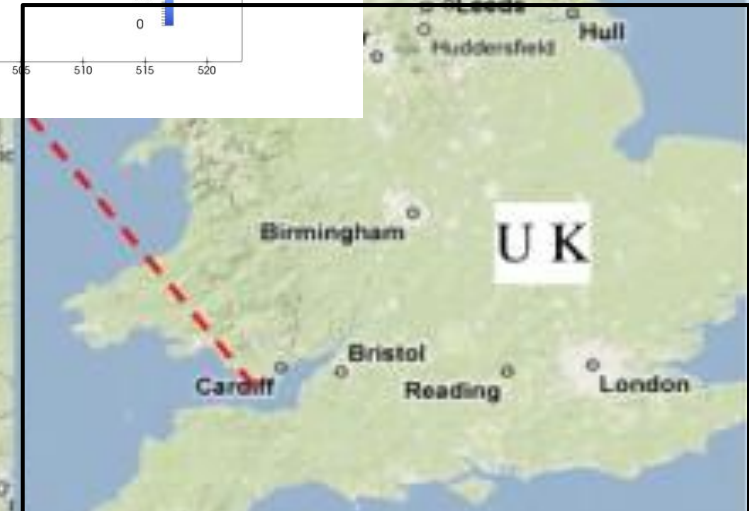
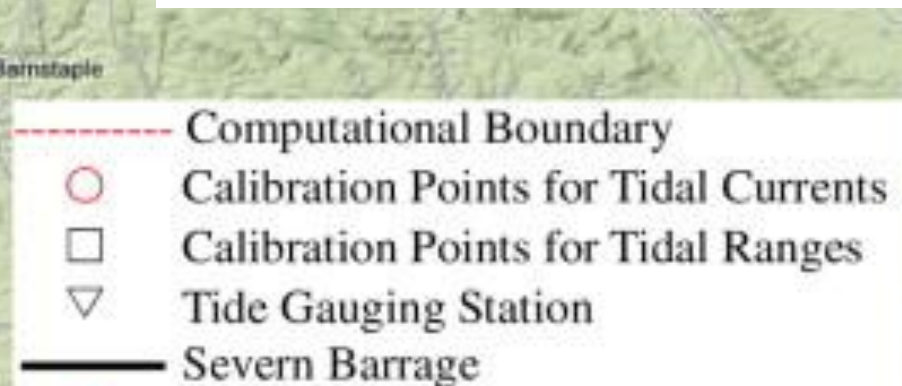
- Estuary of the River Severn: huge tidal energy opportunity
- Significant causes for concern over ecological impact
- Should we do it? How? Where? How much energy? How much impact?



- Estuary of the River Severn: huge tidal energy opportunity
- Significant causes for concern over ecological impact
- Should we do it? How? Where? How much energy? How much impact?

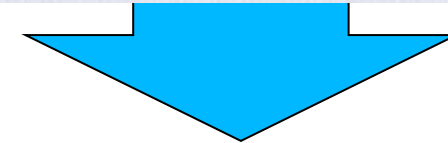
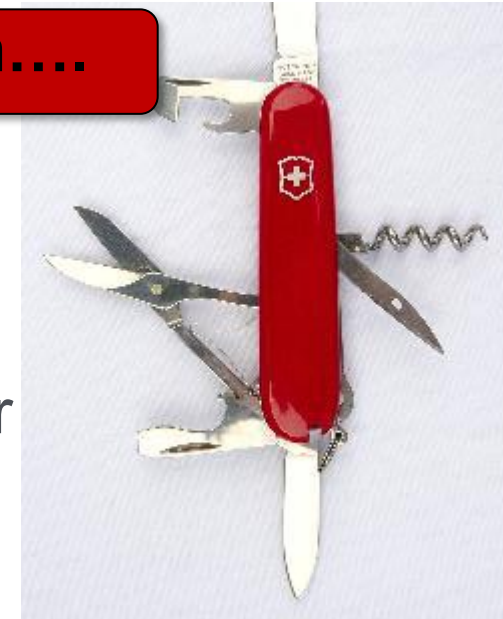


From <https://thetisproject.org/>
 See for example:
 On the potential of linked-basin tidal
 power plants: An operational and
 coastal modelling assessment
 Angeloudis et al *Renewable Energy*
 V155, Aug 2020



By capturing domain-specific representation....

- We can deliver domain-specific **optimisations**
- We collect and automate all the performance techniques that are known for a **family of problems**
- If we get it right.... we get
 - **Productivity** – by generating low-level code from a high-level specification
 - **Performance** – by automating optimisations
 - **Performance portability** – with multiple back-ends



Another example DSL:

<https://www.devitoproject.org/>

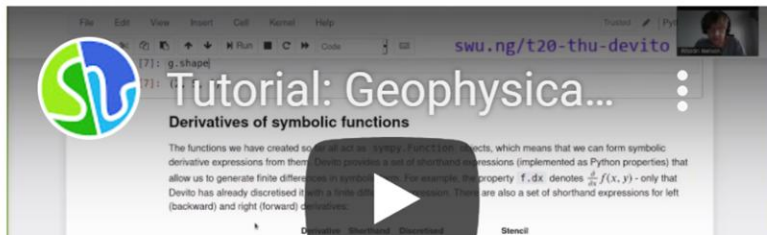
Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hückelheim, J., Yount, C., Kelly, P.H., Herrmann, F.J. & Gorman, G.J. Architecture and performance of Devito, a system for automated stencil computation. ACM Transactions on Mathematical Software (TOMS), 46(1) 2020

Devito: Symbolic Finite Difference Computation

Devito is a domain-specific Language (DSL) and code generation framework for the design of highly optimised finite difference kernels for use in inversion methods. Devito utilises [SymPy](#) to allow the definition of operators from high-level symbolic equations and generates optimised and automatically tuned code specific to a given target architecture.

Symbolic computation is a powerful tool that allows users to:

- Build complex solvers from only a few lines of high-level code
- Use automated performance optimisation for generated code
- Adjust stencil discretisation at runtime as required
- (Re-)development of solver code in hours rather than months



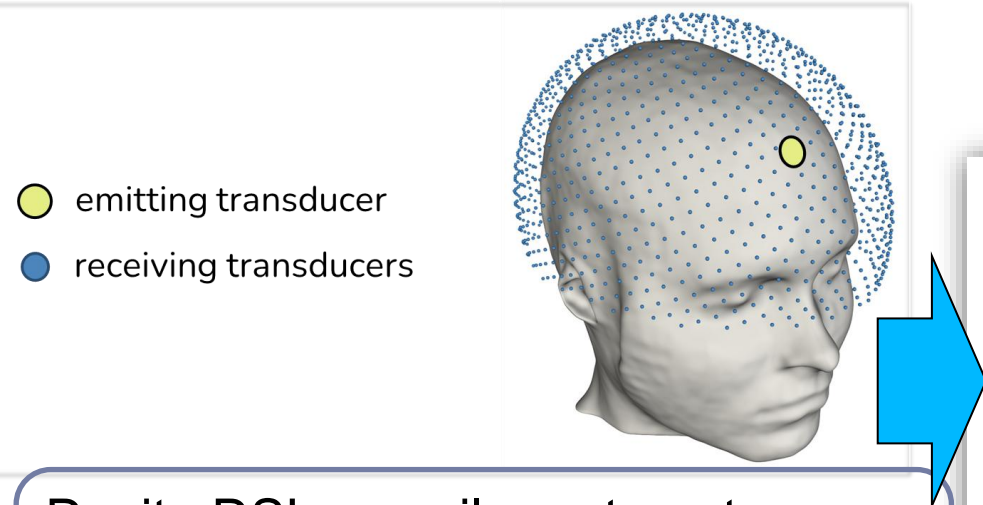
Gerard Gorman Fabio Luporini

And many many more!

- Devito automates the finite difference method for solving PDEs
- Widely used for fluid dynamics, wave propagation
- Devito is mostly used to solve inversion problems
 - Use automatic differentiation of the solver
 - To solve for the conditions that explain the observations
 - “Full Waveform Inversion” (FWI)
- **Seismic inversion**
 - Understand geological structures from reflected sound waves
- **Ultrasound imaging of the brain**
 - Diagnose brain injuries from ultrasound transmission

Devito applications largely driven by seismic inversion

■ Ultrasound imaging of the brain through the skull



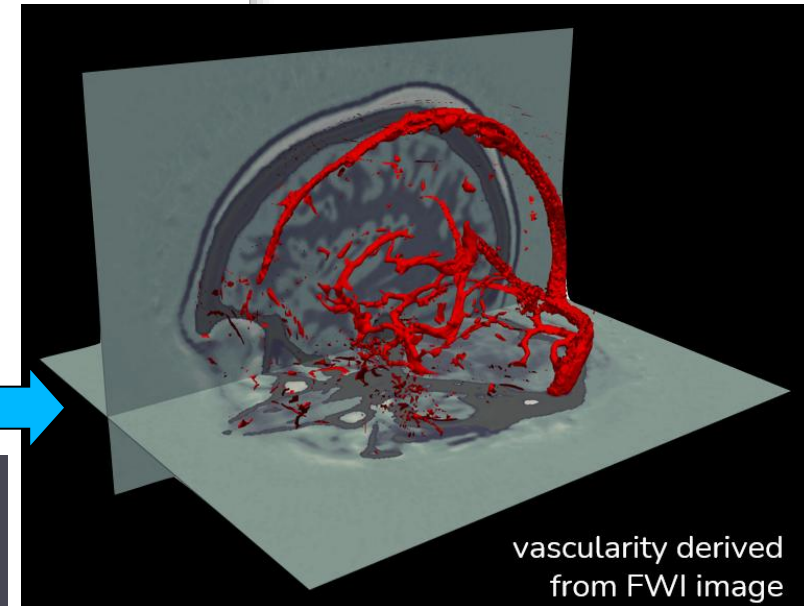
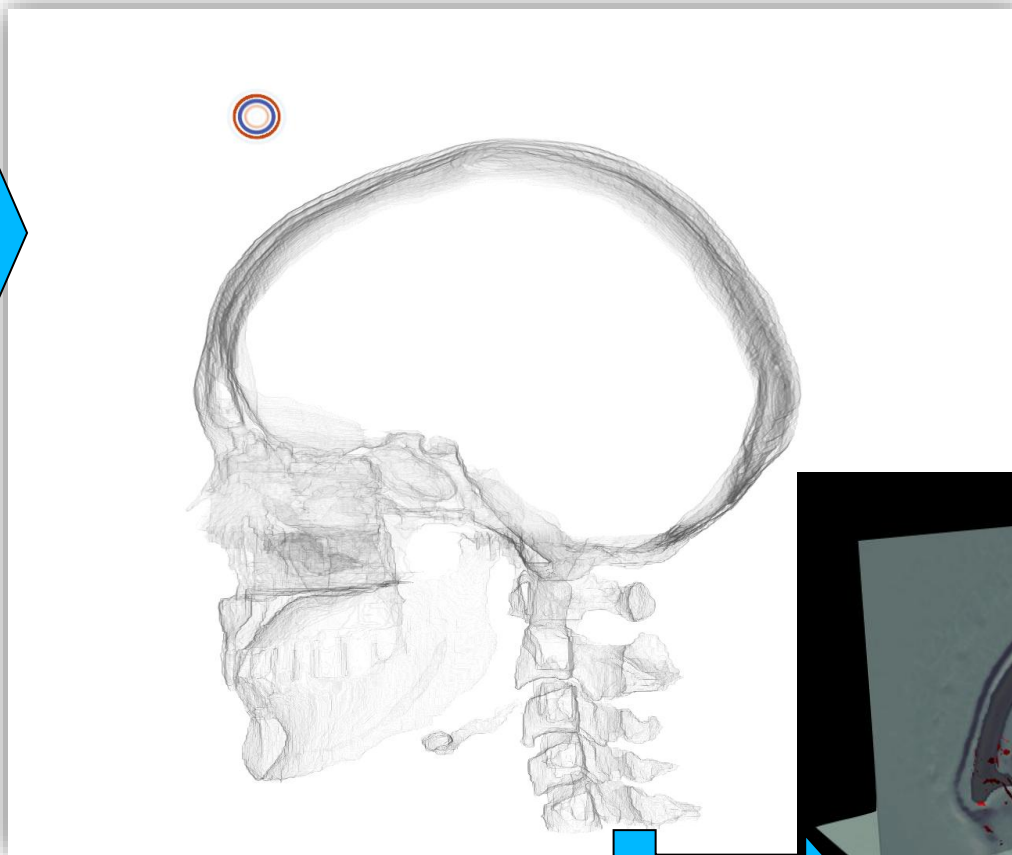
Devito DSL compiler automates
pathway from PDE to high-
performance code

To derive forward wave propagator

And reverse adjoint wave propagator

To compute gradient

With which to correct brain model



Cueto, C., Guasch, L., Loporini, F., Bates, O., Strong, G., Agudo, O.C., Cudeiro, J., Kelly, P., Gorman, G. and Tang, M.X., 2022, April. Tomographic ultrasound modelling and imaging with Stride and Devito. In Medical Imaging 2022: Ultrasonic Imaging and Tomography (p. PC1203805).

Devito: example

Define the wavefield from model setup.

```
u = TimeFunc(time_order=2, space_order=2)
```

Write down the acoustic wave PDE:

```
pde = model.m*u.dt2 - u.laplace + model.damp*u.dt
```

Solve by time-marching:

```
stencil = Eq(u.forward, solve(pde, u.forward))
```

Define source injection and receiver:

```
src_term = src.inject(field=u.forward, pr=src*dt**2/model.m)
```

```
rec_term = rec.interpolate(expr=u.forward)
```

Generate code for the timestepping operator:

```
op = Operator([stencil] + src_term + rec_term,
              subs=model.spacing_map)
```

Run code (MPI+GPU), to yield receiver values:

```
op(time=time_range.num-1, dt=model.critical_dt)
```

Acoustic wave equation, with damping:

$$\begin{cases} m \frac{d^2 u(x,t)}{dt^2} - \nabla^2 u(x,t) + \eta \frac{du(x,t)}{dt} = q \text{ in } \Omega \\ u(\cdot, 0) = 0 \\ \frac{du(x,t)}{dt} \big|_{t=0} = 0 \end{cases}$$

We inject initial sound wave at source point, and monitor the signal at a receiver.

We derive and generate the stencil operator code, then run it a specified number of timesteps

Slightly simplified from:

https://slimgroup.github.io/Devito-Examples/tutorials/01_modelling/

Devito: example

Define the wavefield from model setup.

```
u = TimeFunc(time_order=2, space_order=2)
```

Write down the acoustic wave PDE:

```
pde = model.m*u.dt2 - u.laplace + model.damp*u.dt
```

Solve by time-marching:

```
stencil = Eq(u.forward, solve(pde, u.forward))
```

Define source injection and receiver:

```
src_term = src.inject(field=u.forward, pr=src*dt**2/model.m)
```

```
rec_term = rec.interpolate(expr=u.forward)
```

Generate code for the timestepping operator:

```
op = Operator([stencil] + src_term + rec_term,
              subs=model.spacing_map)
```

Run code (MPI+GPU), to yield receiver values:

```
op(time=time_range.num-1, dt=model.critical_dt)
```

Acoustic wave equation, with damping:

$$\begin{cases} m \frac{d^2 u(x,t)}{dt^2} - \nabla^2 u(x,t) + \eta \frac{du(x,t)}{dt} = q \text{ in } \Omega \\ u(\cdot, 0) = 0 \\ \frac{du(x,t)}{dt} \big|_{t=0} = 0 \end{cases}$$

No-MPI

```
$ python myscript.py
```

With-MPI (2 ranks)

```
$ DEVITO_MPI=basic mpirun -n 2 python myscript.py
```

MPI + GPU ready

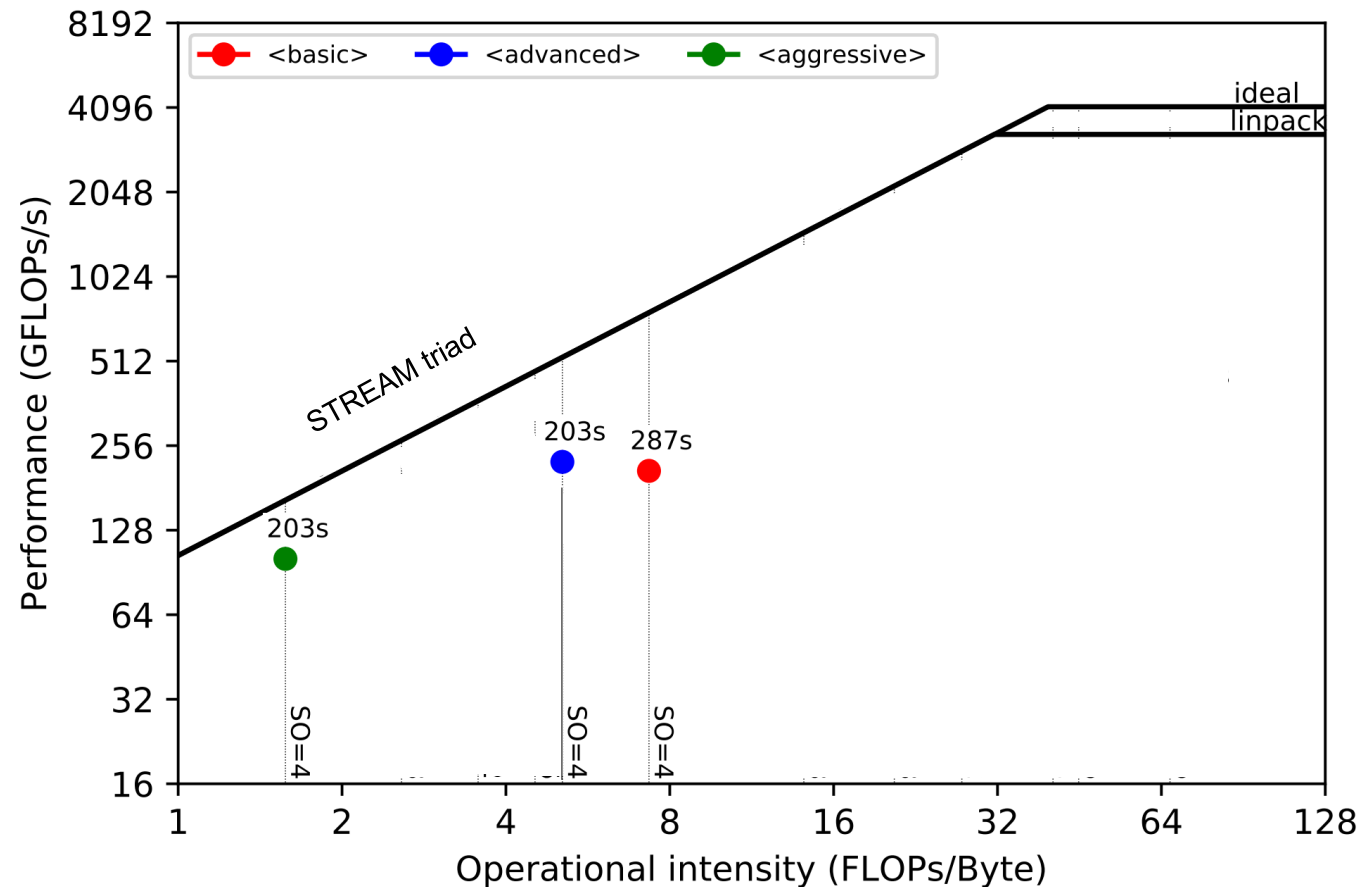
```
# ...add DEVITO_PLATFORM=nvidia DEVITO_COMPILER=nvc
```

Code at this basic level of abstraction is in production, at scale, running at multiple petaflops 24/7

Slightly simplified from:

https://slimgroup.github.io/Devito-Examples/tutorials/01_modelling/

Devito: FLOP-reduction optimisations



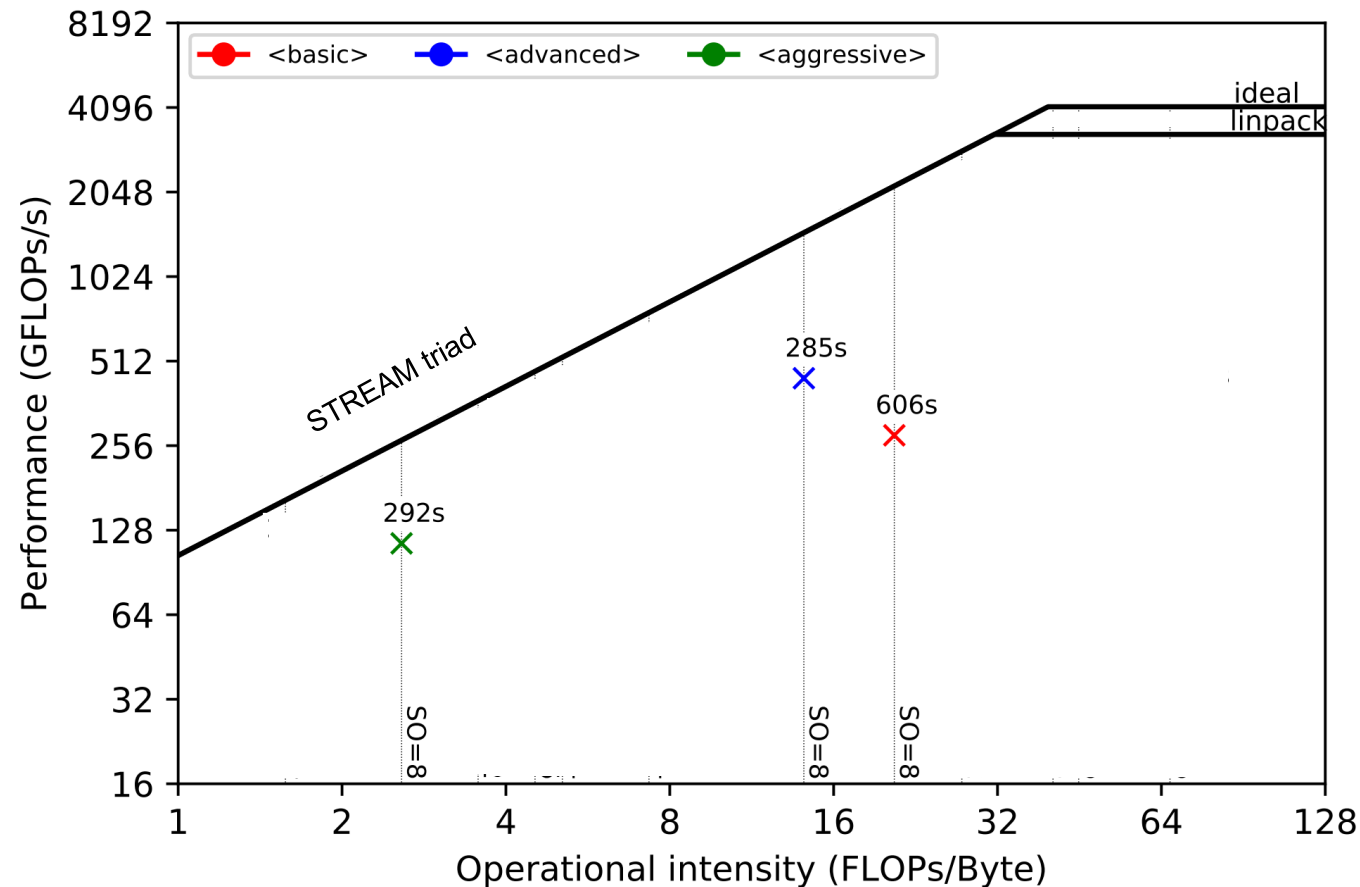
Intel® Xeon® Platinum
8180 (Skylake, 28
cores), ICC v18.0,
Devito v3.1

TTI (Tilted Transverse
Isotropy), second order
in time. 415 timesteps
(1000ms), single
precision.

Space order:
4 (circles)
8 (crosses)
12 (triangles)
16 (nablas)

*Fabio Luporini et al. Architecture and Performance of
Devito, a System for Automated Stencil Computation.
ACM Trans. Math. Softw. 46, 1, Article 6 (April 2020),
<https://doi.org/10.1145/3374916>*

Devito: FLOP-reduction optimisations



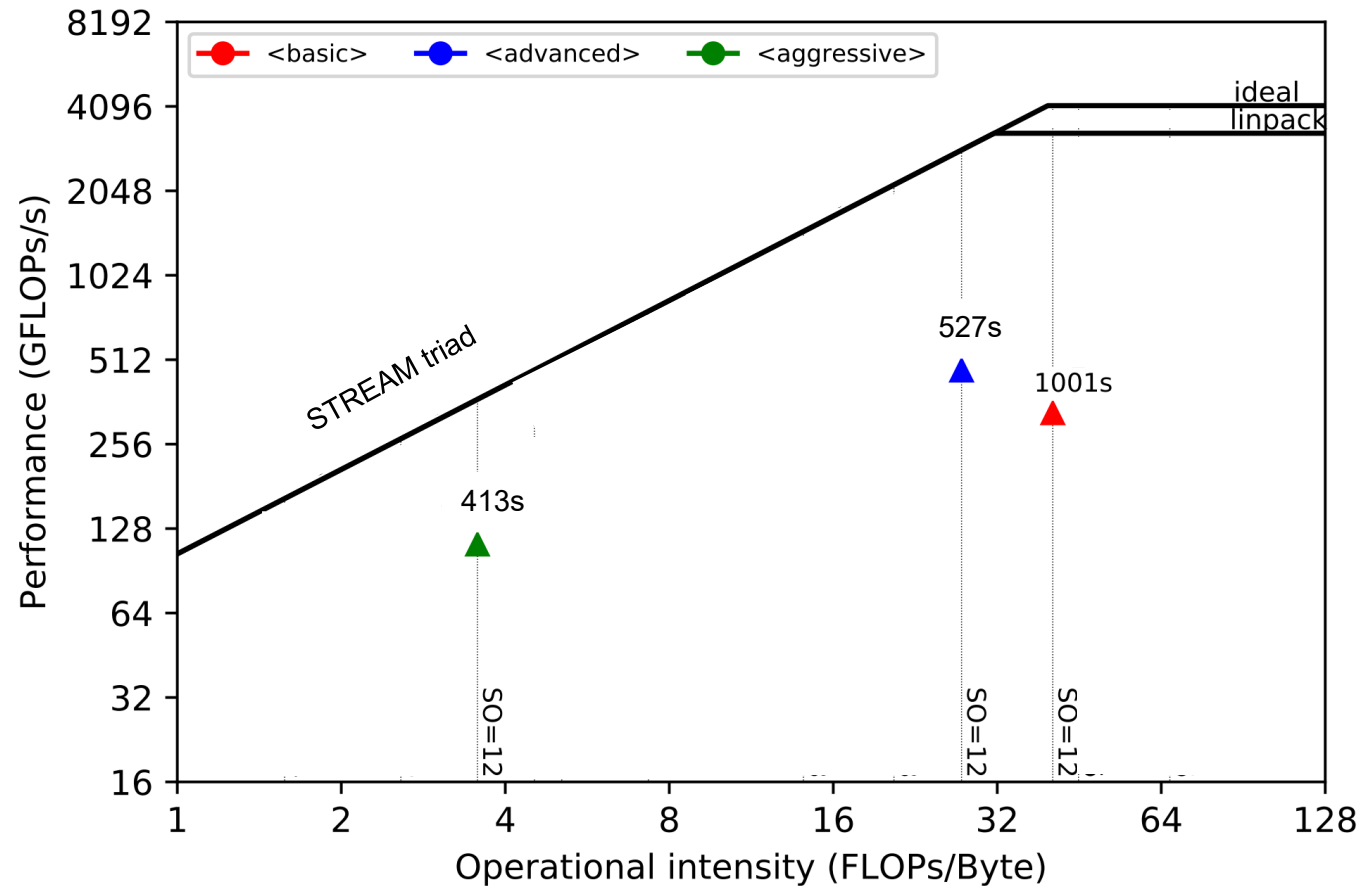
Intel® Xeon® Platinum
8180 (Skylake, 28
cores), ICC v18.0,
Devito v3.1

TTI (Tilted Transverse
Isotropy), second order
in time. 415 timesteps
(1000ms), single
precision.

Space order:
4 (circles)
8 (crosses)
12 (triangles)
16 (nablas)

*Fabio Luporini et al. Architecture and Performance of
Devito, a System for Automated Stencil Computation.
ACM Trans. Math. Softw. 46, 1, Article 6 (April 2020),
<https://doi.org/10.1145/3374916>*

Devito: FLOP-reduction optimisations



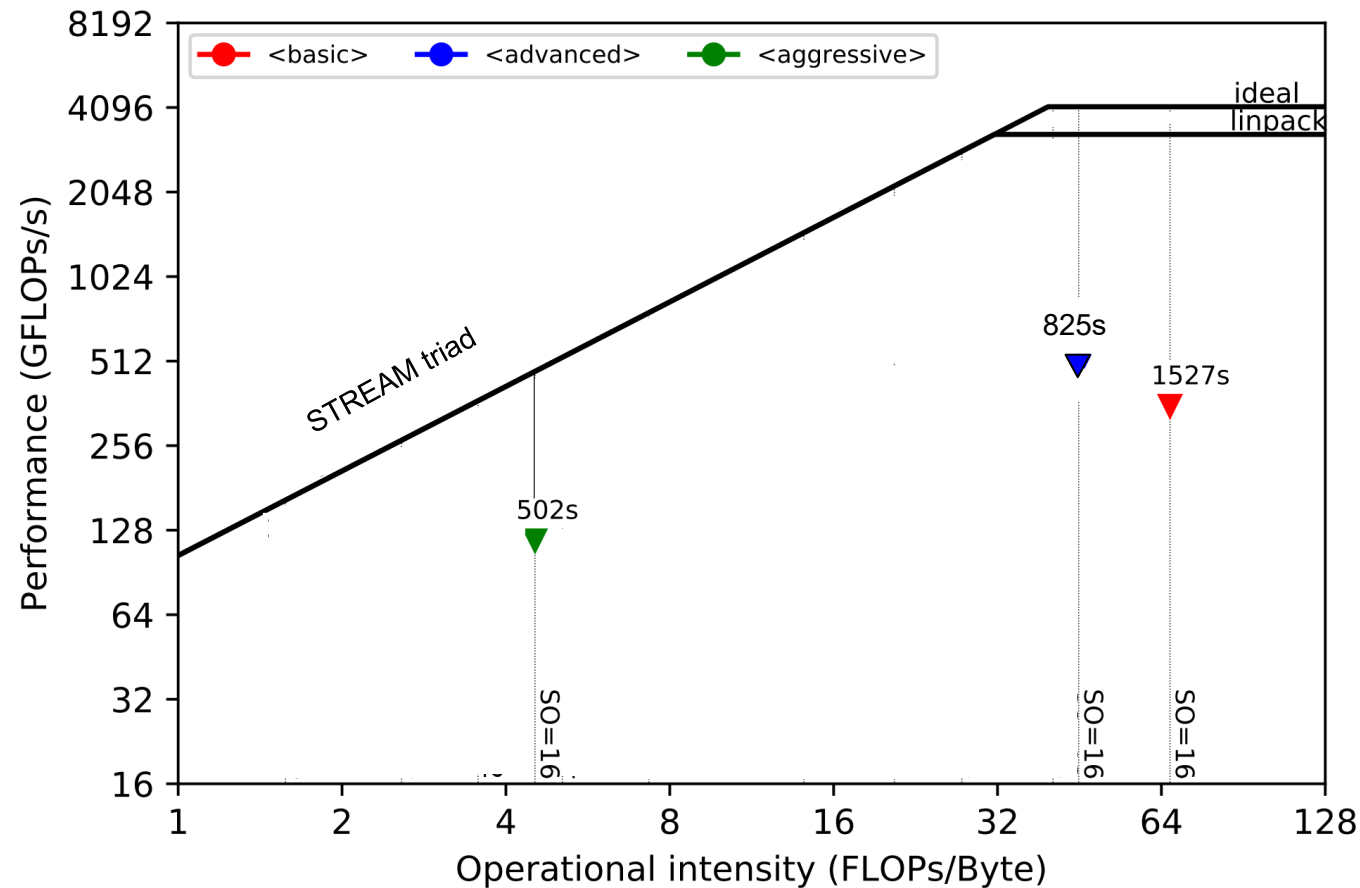
Intel® Xeon® Platinum
8180 (Skylake, 28
cores), ICC v18.0,
Devito v3.1

TTI (Tilted Transverse
Isotropy), second order
in time. 415 timesteps
(1000ms), single
precision.

Space order:
4 (circles)
8 (crosses)
12 (triangles)
16 (nablas)

*Fabio Luporini et al. Architecture and Performance of
Devito, a System for Automated Stencil Computation.
ACM Trans. Math. Softw. 46, 1, Article 6 (April 2020),
<https://doi.org/10.1145/3374916>*

Devito: FLOP-reduction optimisations



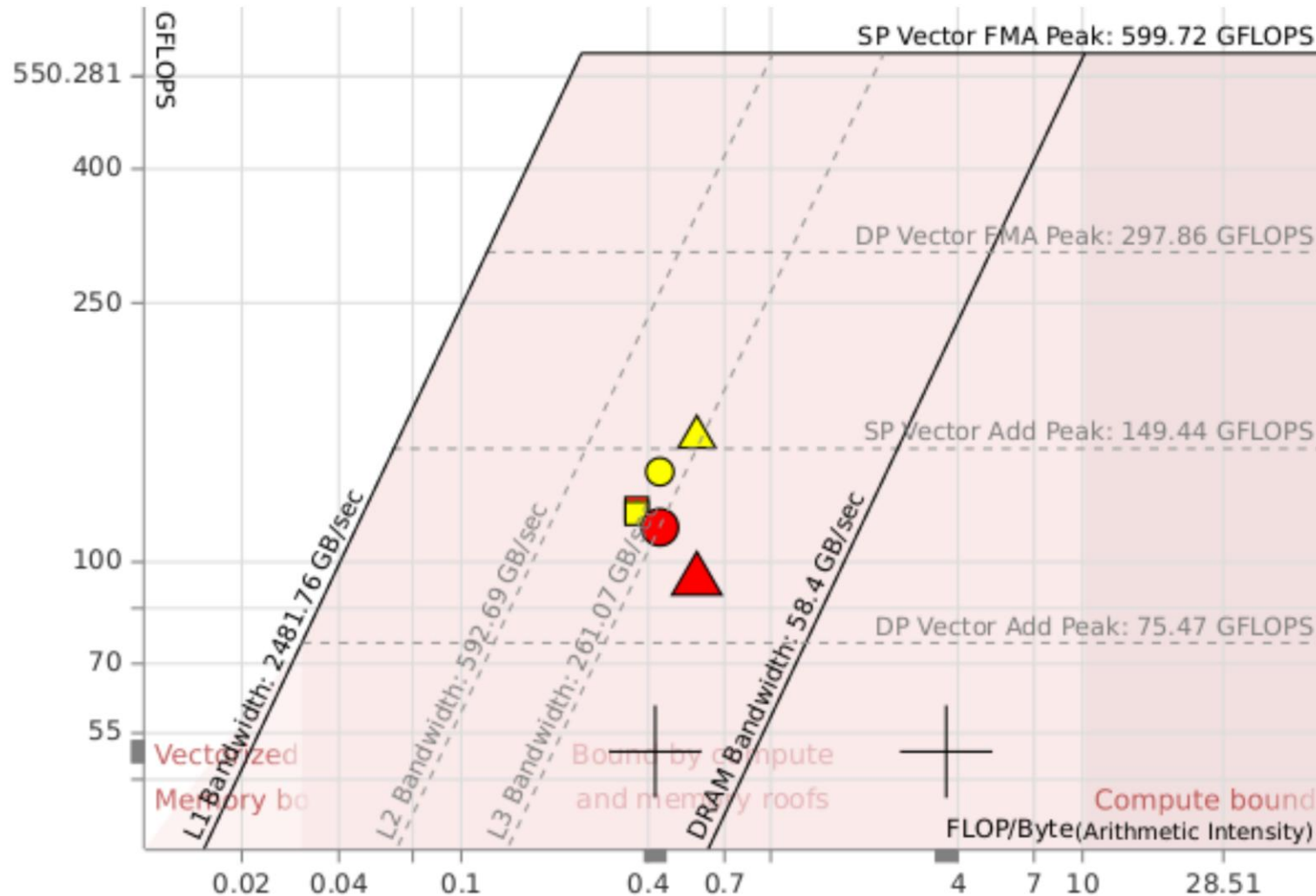
Intel® Xeon® Platinum
8180 (Skylake, 28
cores), ICC v18.0,
Devito v3.1

TTI (Tilted Transverse
Isotropy), second order
in time. 415 timesteps
(1000ms), single
precision.

Space order:
4 (circles)
8 (crosses)
12 (triangles)
16 (nablas)

Fabio Luporini et al. Architecture and Performance of Devito, a System for Automated Stencil Computation. ACM Trans. Math. Softw. 46, 1, Article 6 (April 2020), <https://doi.org/10.1145/3374916>

Devito: tiling-in-time



Single-socket 8-core Intel Broadwell E5-2673 v4 CPUs with AVX2, L1 (32KB), L2 (256KB) private to each core, 50MB shared L3 (Ubuntu 18.04.4, Devito v4.2.3)

Isotropic acoustic model, second-order in time, single-precision

Space order:

4 (triangles), ▲ ▲
 8 (circles), and ● ●
 12 (squares). ■ ■

Red markers show the performance of **spatially** blocked vectorized kernels

Yellow markers show spatial and **temporal** blocking using autotuned tile parameters.

George Bisbas, et al. Temporal blocking of finite-difference stencil operators with sparse “off-the-grid” sources. IPDPS 21
[arXiv:2010.10248](https://arxiv.org/abs/2010.10248)

Fixing the DSL ecosystem

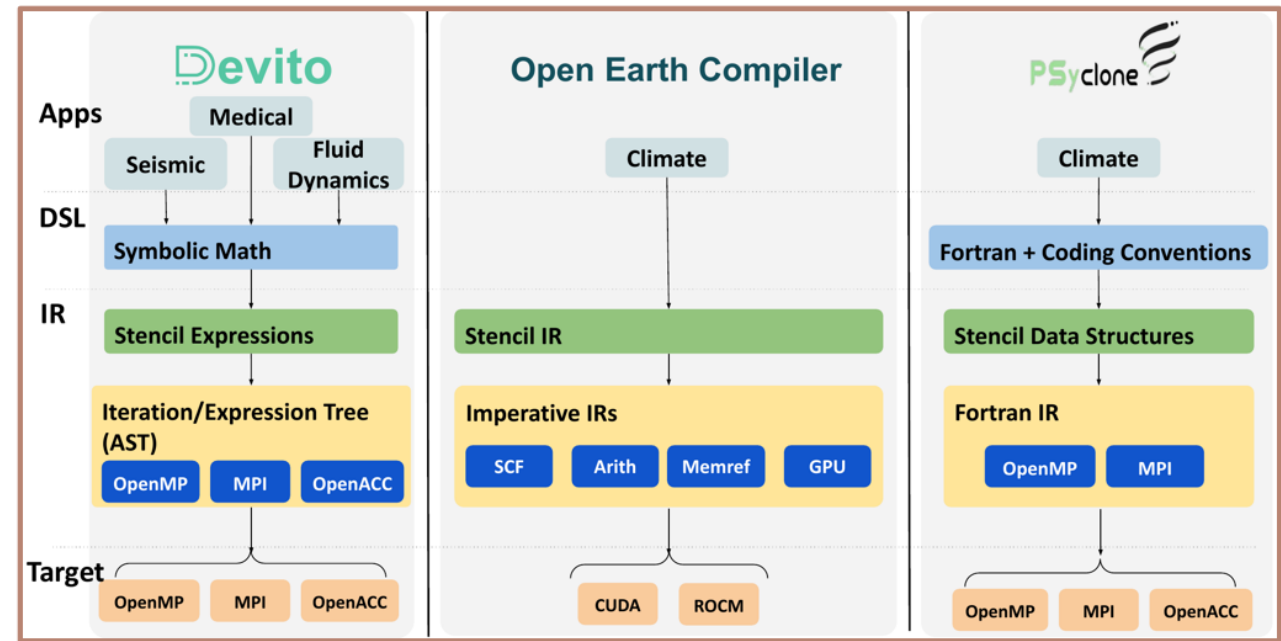
DSLs – domain-specific code generation tools – are expensive to maintain

So we have been exploring how to build on common infrastructure

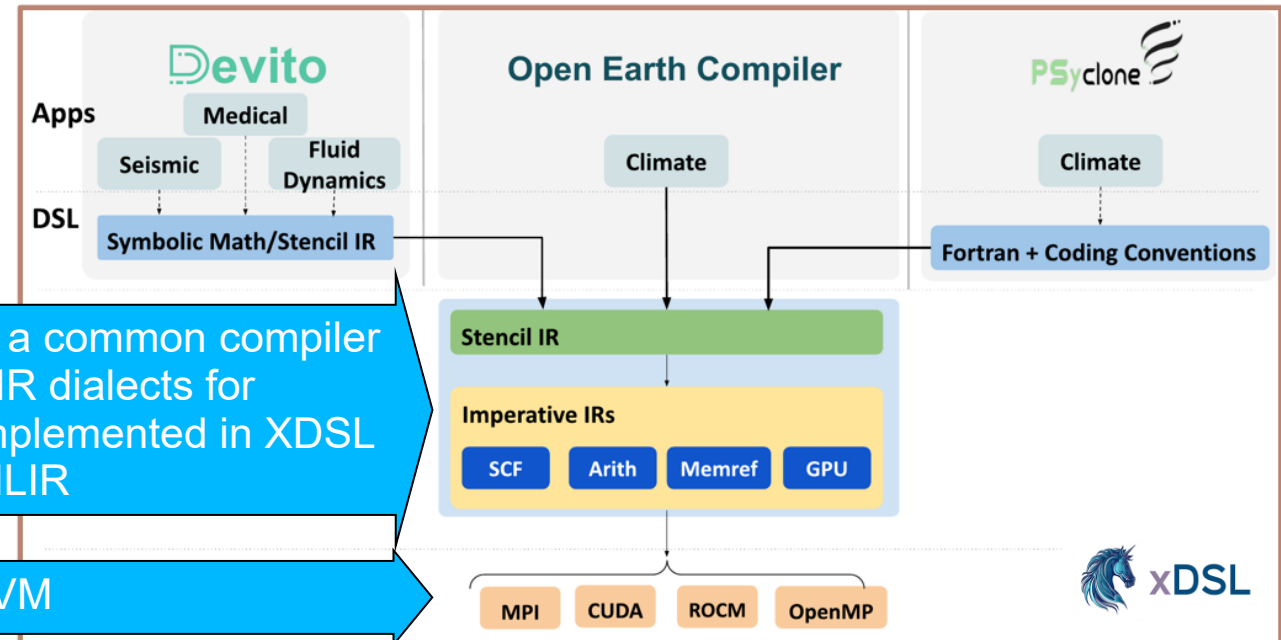
In this work we restructure three stencil DSLs to use common MLIR dialects and transformations

<https://xdsl.dev/>

A shared compilation stack for distributed-memory parallelism in stencil DSLs
Bisbas et al ASPLOS 2024



Three different DSLs each a separate silo



Three DSLs mapping into a common compiler architecture based on MLIR dialects for stencils, MPI, GPU etc, implemented in XDSL – a Python rendering of MLIR

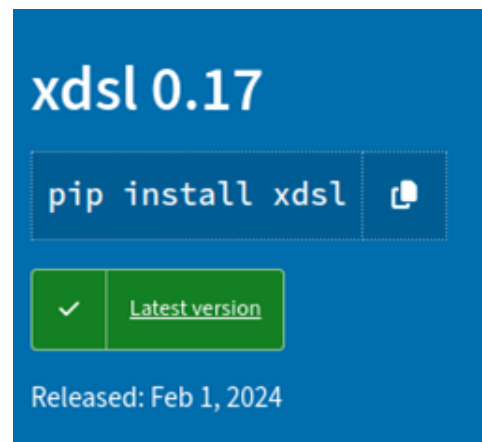
Lowered to MLIR then LLVM



xDSL: A Python-native SSA Compiler Framework

- ✓ SSA-based IRs
- ✓ SSA + regions concept
- ✓ Mix predefined IRs
- ✓ Add custom IRs
- ✓ Connect with MLIR/LLVM
- ✓ Benefit from Python's productivity
- ✓ Open-source/CI/CD/codecov
- ✓ Active contributor community
- ✓ Join us on <https://xdsl.zulipchat.com/>

37



About

A Python Compiler Design Toolkit

Readme

View license

Activity

Custom properties

192 stars

18 watching

54 forks

Report repository

Releases 26

v0.17 **Latest**
on Feb 1

+ 25 releases

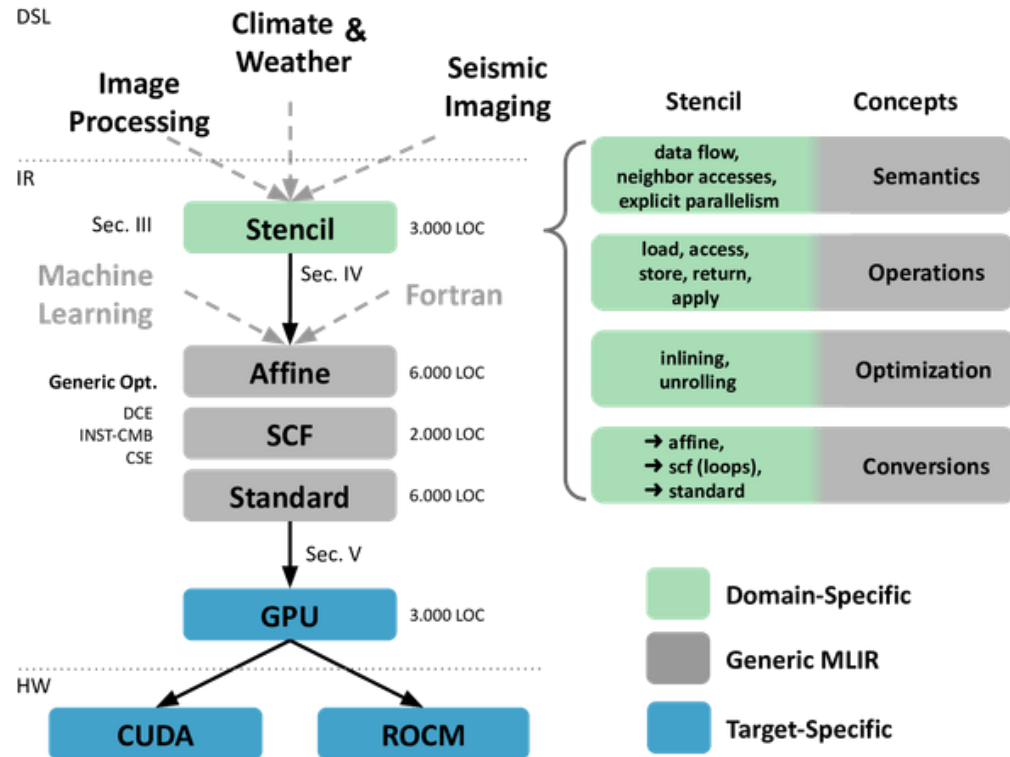
Contributors 54



+ 40 contributors

Fehr, Weber, Ulmann, Lopoukhine, Lücke, Degioanni, Vasiladiotis, Steuer, and Grosser. 2025. XDSL: Sidekick Compilation for SSA-Based Compilers. CGO'25

The Open Earth Compiler: the ‘stencil’ dialect



```
1 %source = stencil.load(%114) : (!field<[0,128]xf64>)  
2   -> !temp<?xf64>  
3 %out = stencil.apply(%arg = %source : !temp<?xf64>)  
4   -> !temp<?xf64> {  
5   %l = stencil.access %arg[-1] : f64  
6   %c = stencil.access %arg[0] : f64  
7   %r = stencil.access %arg[1] : f64  
8   // %v = %l + %r - 2.0 * %c  
9   stencil.return %v : f64  
10 }  
11  
12 stencil.store %out to %target([1]:[127])  
13   : !temp<?xf64> to !field<[0,128]xf64>
```

Listing 1. Example MLIR for 1-dimensional 3-point Jacobi stencil.

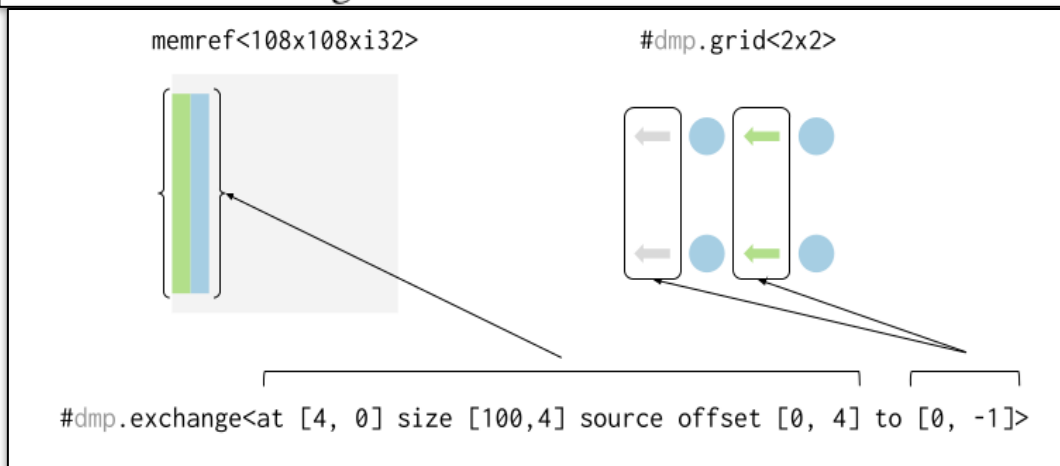
- ✓ Updated, ported to xDSL
- ✓ Extended to multi-node

Gysi et.al, Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate (2021), ACM TACO
<https://github.com/spcl/open-earth-compiler/>

The 'dmp' and 'mpi' dialects

```
1 dmp.swap(%data)
2 {
3   "grid" = #dmp.grid<2x2>,
4   "swaps" = [
5     #dmp.exchange<at [4, 0] size [100, 4]
6       source offset [0, 4] to [0, -1]>,
7     #dmp.exchange<at [4, 104] size [100, 4]
8       source offset [0, -4] to [0, 1]>
9   ]
10 } : (memref<108x108xf32>) -> ()
```

Listing 2. A high-level declarative expression of a data subsection exchange from some buffer.



- ✓ High-level halo exchanges
- ✓ Describe communication patterns
- ✓ Rectangular data subsections

'mpi' Dialect

This dialect models the Message Passing Interface (MPI), version 4.0. It is meant to serve as an interfacing dialect that is targeted by higher-level dialects. The MPI dialect itself can be lowered to multiple MPI implementations and hide differences in ABI. The dialect models the functions of the MPI specification as close to 1:1 as possible while preserving SSA value semantics where it makes sense, and uses `memref` types instead of bare pointers.

This dialect is under active development, and while stability is an eventual goal, it is not guaranteed at this juncture. Given the early state, it is recommended to inquire further prior to using this dialect.

For an in-depth documentation of the MPI library interface, please refer to official documentation such as the [OpenMPI online documentation](#).

- [Operations](#)
 - `mpi.comm_rank` (`mpi::CommRankOp`)
 - `mpi.error_class` (`mpi::ErrorClassOp`)
 - `mpi.finalize` (`mpi::FinalizeOp`)
 - `mpi.init` (`mpi::InitOp`)
 - `mpi.recv` (`mpi::RecvOp`)
 - `mpi.retval_check` (`mpi::RetValCheckOp`)
 - `mpi.send` (`mpi::SendOp`)
- [Attributes](#)
 - `MPI_ErrorClassEnumAttr`
- [Types](#)
 - `RetValType`

- ✓ Message-passing IR
- ✓ Lowered to MPI library calls
- ✓ Upstreamed to MLIR!

Lowering from 'stencil' to 'mpi'

Stencil level IR

Global to Local

DMP level IR

DMP to MPI

MPI level IR

```
%source = stencil.load(%114) : (!field<[0,128]xf64>)  
  -> !temp<?xf64>  
%out = stencil.apply(%arg = %source : !temp<?xf64>)  
  -> !temp<?xf64> {  
  %l = stencil.access %arg[-1] : f64  
  %c = stencil.access %arg[0] : f64  
  %r = stencil.access %arg[1] : f64  
  // %v = %l + %r - 2.0 * %c  
  stencil.return %v : f64  
}  
  
stencil.store %out to %target([1]:[127])
```



Global Domain

```
%ref = builtin.unrealized_conversion_cast %114 :  
  !field<[0,64]xf64> to memref<64xf62>  
dmp.swap(%ref) {  
  "grid" = #dmp.grid<2>,  
  "swaps" = [  
    #dmp.exchange<at [0] size [1]  
      source offset [1] to [-1]>,  
    #dmp.exchange<at [64] size [1]  
      source offset [-1] to [1]>  
  ]  
} : (memref<64xf64>) -> ()  
%source = stencil.load(%114) ...  
%out = stencil.apply(%source) ...  
stencil.store %out to %target([1]:[64])
```



Local Domains with halo exchanges highlighted

```
%rank = mpi.comm_rank : i32  
// First swap communication calls  
%dest = arith.add %rank, %minus_one : i32  
%is_in_bounds = arith.cmpi sge, %dest, %zero  
scf.if %is_in_bounds {  
  %view = memref.subview %ref[0][1][1] : memref<64xf64>  
    to memref<1xf64>  
  // copy data into send buffer and set up communication  
  // (omitted for clarity)  
  mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req  
  mpi.irecv %rptr, %count, %dtype, %dest, %tag, %recv_req  
}  
// Second swap  
// ...  
mpi.waitall %requests, %four // synchronization barrier  
// First swap copy back  
scf.if %is_in_bounds {  
  %view = memref.subview %ref[1][1][1] : memref<64xf64>  
    to memref<1xf64>  
  memref.copy %recv_buffer_1 to %view  
}  
// Second swap copy back  
// Lowered stencil comes here
```

Colours highlight data being operated on, shape and halo information, and communication-related information. This shows how we can enrich the IR with relevant information to perform efficient rewrites at every level of abstraction.

Lowering from 'stencil' to 'mpi'

Stencil level IR

Global to Local

```
%source = stencil.load(%114) : (!field<[0,128]xf64>)  
      -> !temp<?xf64>  
%out = stencil.apply(%arg = %source : !temp<?xf64>)  
      -> !temp<?xf64> {  
  %l = stencil.access %arg[-1] : f64  
  %c = stencil.access %arg[0] : f64  
  %r = stencil.access %arg[1] : f64  
  // %v = %l + %r - 2.0 * %c  
  stencil.return %v : f64  
}  
stencil.store %out to %target([1]:[127])
```

1

127

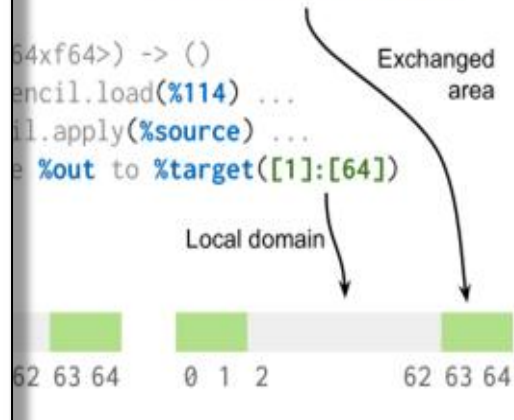
Global Domain

DMP level IR

DMP to MPI

MPI level IR

```
in.unrealized_conversion_cast %114 :  
[0,64]xf64> to memref<64xf62>  
{  
  dmp.grid<2>,  
  change<at [0] size [1]  
    source offset [1] to [-1]>,  
  change<at [64] size [1]  
    source offset [-1] to [1]>  
  64xf64>) -> ()  
  stencil.load(%114) ...  
  stencil.apply(%source) ...  
  stencil.store %out to %target([1]:[64])  
}
```



Exchanged area

Local domain

62 63 64 0 1 2 62 63 64

Domains with halo exchanges highlighted

```
%rank = mpi.comm_rank : i32  
// First swap communication calls  
%dest = arith.add %rank, %minus_one : i32  
%is_in_bounds = arith.cmpi sge, %dest, %zero  
scf.if %is_in_bounds {  
  %view = memref.subview %ref[0][1][1] : memref<64xf64>  
    to memref<1xf64>  
  // copy data into send buffer and set up communication  
  // (omitted for clarity)  
  mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req  
  mpi.irecv %rptr, %count, %dtype, %dest, %tag, %recv_req  
}  
// Second swap  
// ...  
mpi.waitall %requests, %four // synchronization barrier  
// First swap copy back  
scf.if %is_in_bounds {  
  %view = memref.subview %ref[1][1][1] : memref<64xf64>  
    to memref<1xf64>  
  memref.copy %recv_buffer_1 to %view  
}  
// Second swap copy back  
// Lowered stencil comes here
```

Colours highlight data being operated on, shape and halo information, and communication-related information. This shows how we can enrich the IR with relevant information to perform efficient rewrites at every level of abstraction.

Lowering from 'stencil' to 'mpi'

Stencil level IR

```
%source = stencil.load(%114) : (!field, f64) -> !tensors[1]xf64
%out = stencil.apply(%arg = %source, %temp = !temp[?xf64] {
  %l = stencil.access %arg[-1] : f64
  %c = stencil.access %arg[0] : f64
  %r = stencil.access %arg[1] : f64
  // %v = %l + %r - 2.0 * %c
  stencil.return %v : f64
})
stencil.store %out to %target([1]:[128])
```

1

Global Domain

Global to Local

DMP level IR

```
%ref = builtin.unrealized_conversion_cast %114 : (!field, f64) -> !memref<64xf64>
dmp.swap(%ref) {
  "grid" = #dmp.grid<2>,
  "swaps" = [
    #dmp.exchange<at [0] size [1]
      source offset [1] to [-1]>,
    #dmp.exchange<at [64] size [1]
      source offset [-1] to [1]>
  ]
} : (memref<64xf64>) -> ()
%source = stencil.load(%114) ...
%out = stencil.apply(%source) ...
stencil.store %out to %target([1]:[64])
```

Exchanged area

Local domain



Local Domains with halo exchanges highlighted

DMP to MPI

MPI level IR

```
%rank = mpi.comm_rank : i32
// First swap communication calls
%dest = arith.add %rank, %minus_one : i32
%is_in_bounds = arith.cmpi sge, %dest, %zero
scf.if %is_in_bounds {
  %view = memref.subview %ref[0][1][1] : memref<64xf64>
  // copy data into send buffer and set up communication
  // (omitted for clarity)
  mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req
  mpi.irecv %rprr, %count, %dtype, %dest, %tag, %recv_req
}
// Second swap
// ...
mpi.waitall %requests, %four // synchronization barrier
// First swap copy back
scf.if %is_in_bounds {
  %view = memref.subview %ref[1][1][1] : memref<64xf64>
  memref.copy %recv_buffer_1 to %view
}
// Second swap copy back
// Lowered stencil comes here
```

Colours highlight data being operated on, shape and halo information, and communication-related information. This shows how we can enrich the IR with relevant information to perform efficient rewrites at every level of abstraction.

Lowering from 'stencil' to 'mpi'

Stencil level IR

Global to Local

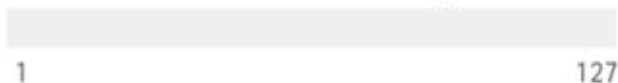
DMP level IR

DMP to MPI

MPI level IR

```
%source = stencil.load(%114) : (!field<[0,128]xf64>
  -> !temp<?xf64>)
%out = stencil.apply(%arg = %source : !temp<?xf64>)
  -> !temp<?xf64> {
  %l = stencil.access %arg[-1] : f64
  %c = stencil.access %arg[0] : f64
  %r = stencil.access %arg[1] : f64
  // %v = %l + %r - 2.0 * %c
  stencil.return %v : f64
}

stencil.store %out to %target([1]:[127])
```



```
%ref = builtin.unrealized
  !field<[0,64]xf64> to
dmp.swap(%ref) {
  "grid" = #dmp.grid<2>,
  "swaps" = [
    #dmp.exchange<at [0]
      source
    #dmp.exchange<at [64]
      source
  ]
} : (memref<64xf64>) -> (
%source = stencil.load(%1
%out = stencil.apply(%sou
stencil.store %out to %ta
```



```
%rank = mpi.comm_rank : i32
// First swap communication calls
%dest = arith.add %rank, %minus_one : i32
%is_in_bounds = arith.cmpi sge, %dest, %zero
scf.if %is_in_bounds {
  %view = memref.subview %ref[0][1][1] : memref<64xf64>
    to memref<1xf64>

  // copy data into send buffer and set up communication
  // (omitted for clarity)
  mpi.isend %sptr, %count, %dtype, %dest, %tag, %send_req
  mpi.irecv %rptr, %count, %dtype, %dest, %tag, %recv_req
}

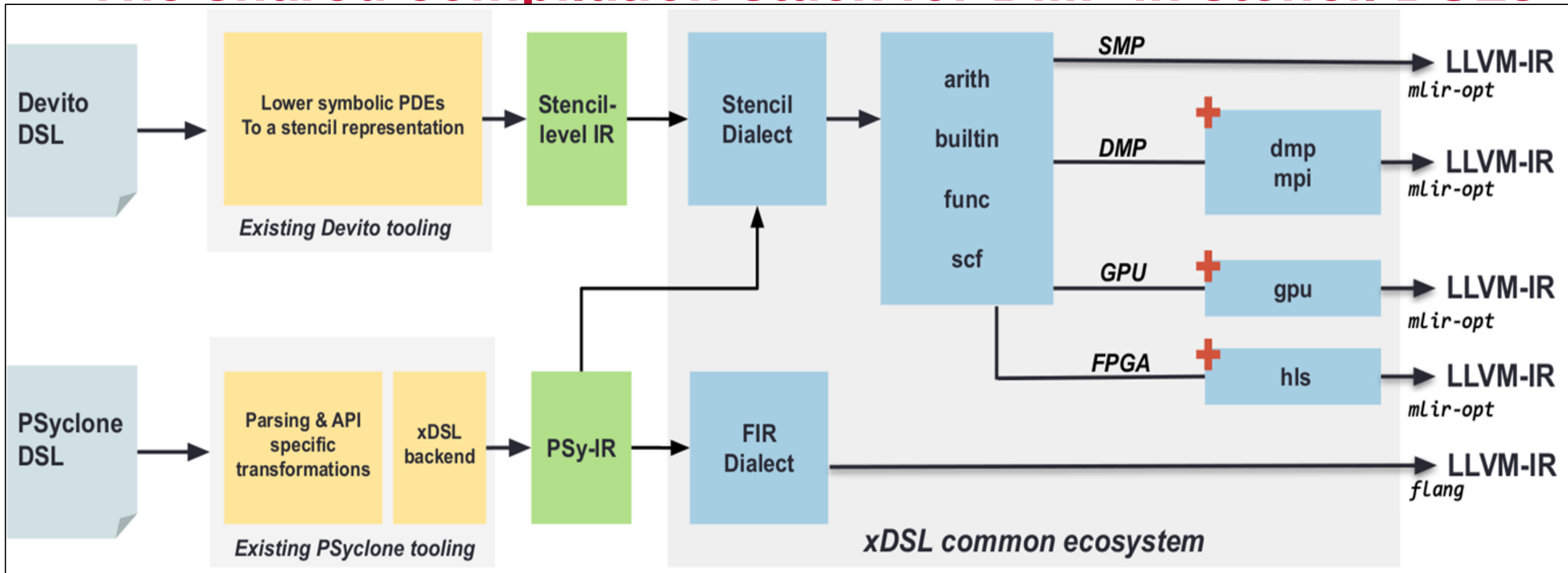
// Second swap
// ...
mpi.waitall %requests, %four // synchronization barrier
// First swap copy back
scf.if %is_in_bounds {
  %view = memref.subview %ref[1][1][1] : memref<64xf64>
    to memref<1xf64>

  memref.copy %recv_buffer_1 to %view
}

// Second swap copy back
// Lowered stencil comes here
```

Colours highlight data being operated on, shape and halo information, and communication-related information. This shows how we can enrich the IR with relevant information to perform efficient rewrites at every level of abstraction.

The shared compilation stack for DMP in stencil DSLs

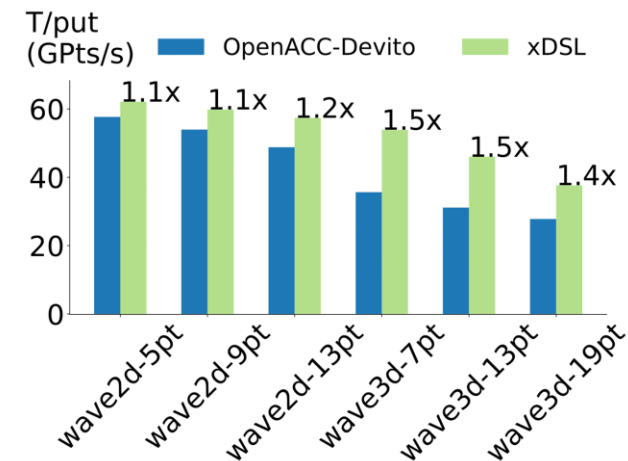
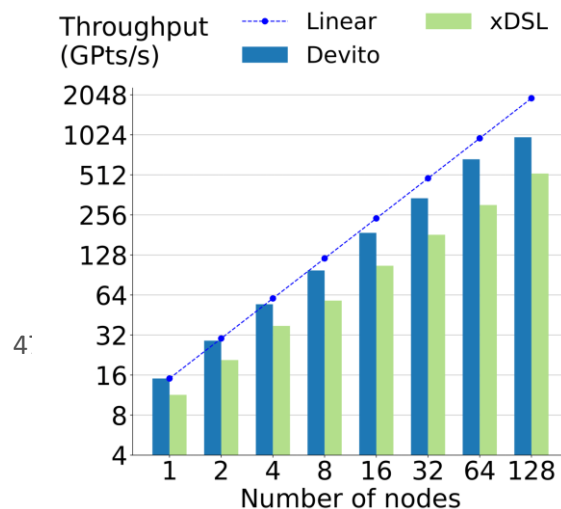
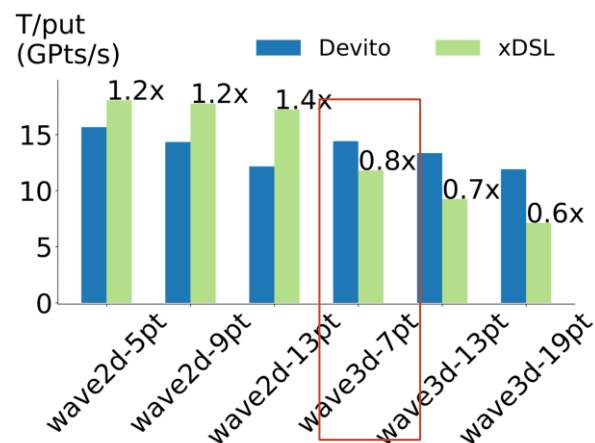
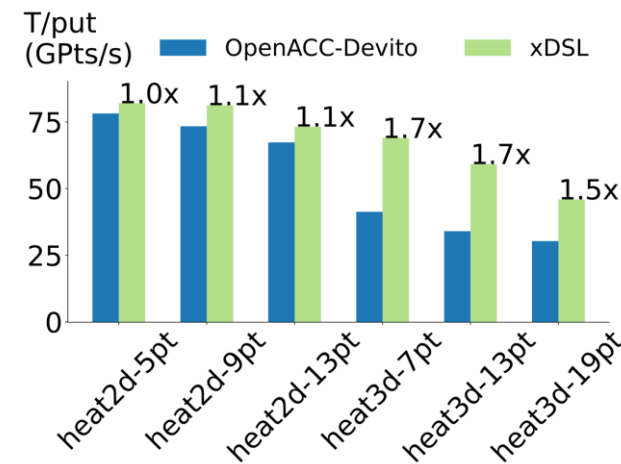
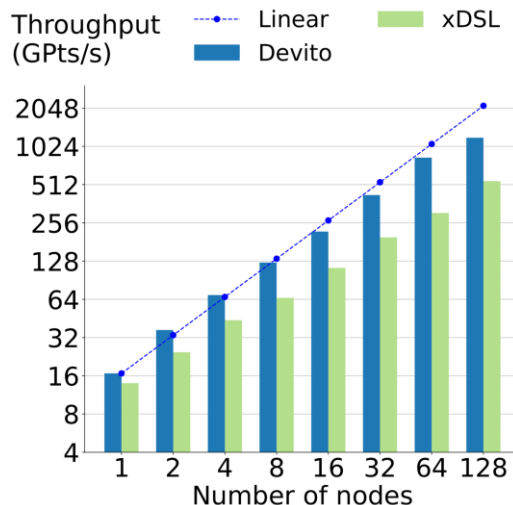
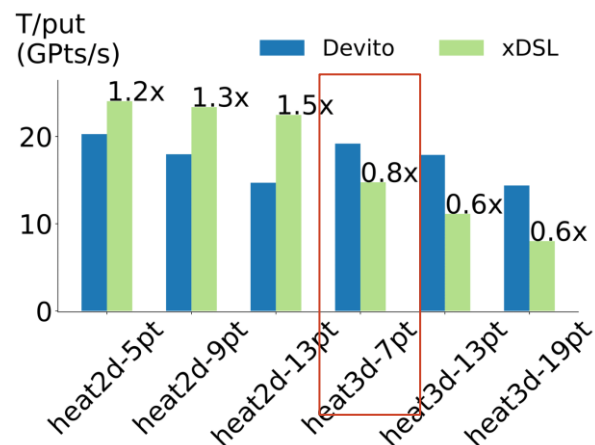


46

- ✓ Unlocked optimizations
- ✓ Unlocked multi-node CPU
- ✓ Unlocked other backends (FPGA, CUDA)

✓ Competitive or better performance with an order of 1000s LoC saved!

Performance evaluation: Devito



Single-node AMD EPYC 7742,
8 MPI ranks x 16 OpenMP threads,
16384² (2D) and 1024³ (3D)

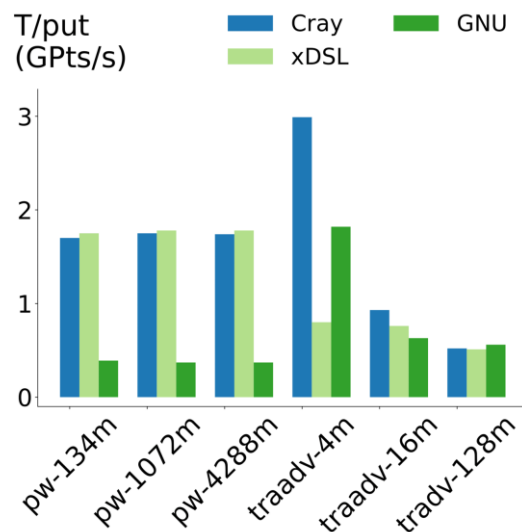
Heat (top) and wave (bottom), 3D-7pt, multi-
node strong scaling up to 128 nodes, total of
16384 cores.

xDSL adds support for CUDA, outperforming
Devito's OSS support for OpenACC, running on
V100-SXM2-16GB (Volta).

Performance evaluation: PSyclone

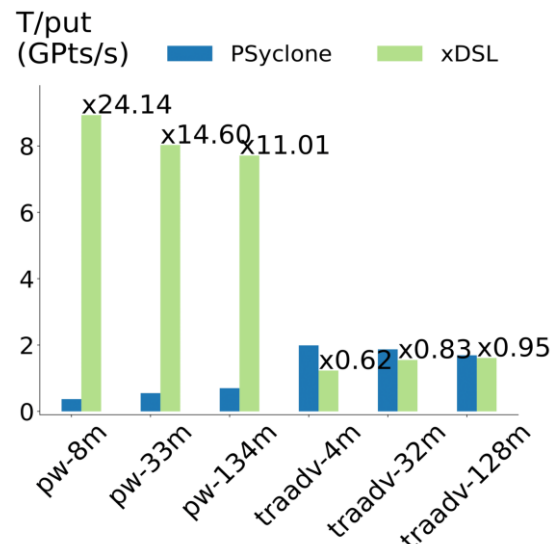


Piacsek and Williams (PW) advection and NEMO tracer advection (traadv) kernels



Single-node AMD EPYC 7742 throughput, PSyclone target code compiled with Cray and GNU compilers against xDSL-PSyclone.

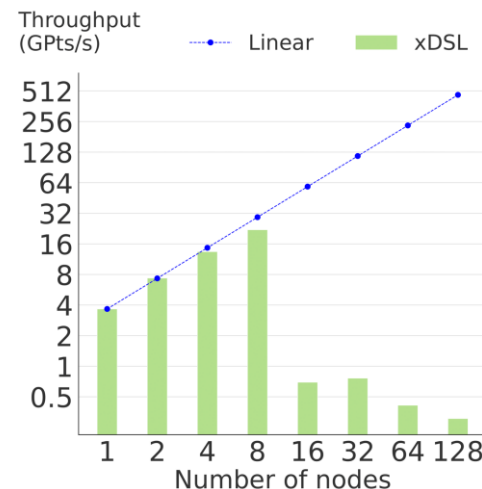
xDSL-PSyclone code matches Cray code and significantly outperforms GNU code for PW advection.



Single-node Cirrus NVIDIA Tesla V100-SXM2-16GB throughput, PSyclone NVIDIA GPU code against xDSL-PSyclone GPU code.

xDSL-PSyclone significantly outperforms PSyclone NVIDIA for PW advection due to data allocation approach (explicit device memory allocation for xDSL-PSyclone).

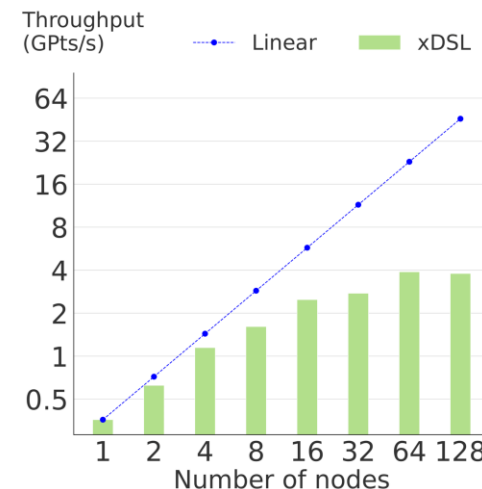
PW advection kernel



Multi-node strong scaling of problem size [256,256,128], scaling up to 128 nodes, total of 16384 cores.

Suffers scaling effects at 8 nodes due to small global problem size.

traadv kernel



Multi-node strong scaling of problem size [512,512,128], scaling up to 128 nodes, total of 16384 cores.

2D decomposition strategy limits strong scaling.

Conclusions

- **Productivity** – by generating low-level code from a high-level specification
- **Performance** – by automating optimisations
- **Performance portability** – with multiple back-ends

- **Domain-specific compiler design is *all* about designing representations that make hard problems easy**

- **The grand project is to build *common compiler infrastructure* that spans different domains**

- **DSL compilers exploit data structures**

- **DSL compilers exploit computation**

- **Redundancy**
- **Locality**
- **Parallelism**

- **You can actually have it all, today**

- **Tensor contractions**
- **Access/execute**

- **With MLIR**
- **Via XDSL**

- **Meshes**
- **Metadata**
- **Taming pointers**
- **Composition**
- **Adaptivity**

- Thank you to:
 - EPSRC
 - EP/Y020499/1 On-Sensor Computer Vision
 - EP/W026066/1 SysGenX: Composable software generation for system-level simulation at Exascale
 - EP/W007789/1 Efficient Cross-Domain DSL Development for Exascale
 - EP/V001493/1 Gen X: ExCALIBUR working group on Exascale continuum mechanics through code generation
 - EP/R029423/1 PRISM: Platform for Research In Simulation Methods
 - EP/P010040/1 Application Customisation: Enhancing Design Quality and Developer Productivity
 - EP/K008730/1 PAMELA: a Panoramic Approach to the Many-CorE Landscape - from end-user to end-device: a holistic game-changing approach
 - EP/I00677X/1 Multi-layered abstractions for PDEs