Imperial College London

Towards cross-domain domain-specific compiler architecture

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

Joint work with David Ham (Imperial Maths), Lawrence Mitchell (University of Durham) George Bisbas, Edward Stow, Fabio Luporini (Devito Codes Ltd), Florian Rathgeber (now with Google), Doru Bercea (now with IBM Research), Michael Lange (now with ECMWF), Andrew McRae (now at University of Oxford), Graham Markall (now at NVIDIA), Tianjiao Sun (now at Cerebras), Thomas Gibson (NCSA Illinois), Kaushik Kulkarni (UIUC), Andreas Klockner (UIUC), Tobias Grosser, Michel Steuwer (University of Edinburgh), Amrey Krause, Nick Brown (EPCC)
And many others....

Imperial College London

Who am I and what do I do?

- I have worked on general-purpose compilers
  - Notably pointer analysis
  - adopted into GCC
    - (actually the work of my PhD student 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

File:Victorinox Swiss Army Knife - Climber (15554551505).jpg - Wikimedia Commons

Imperial College London

Power tools for performance programming

This talk is about **domain-specific languages**

- So we can deliver domain-specific **optimisations**
- So 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

https://commons.wikimedia.org/wiki/File:Victorinox_Swiss_Army_Knife_-_Climber_(15554551505).jpg
https://commons.wikimedia.org/wiki/File:Makita_DJV181.jpg

Have your cake and eat it too

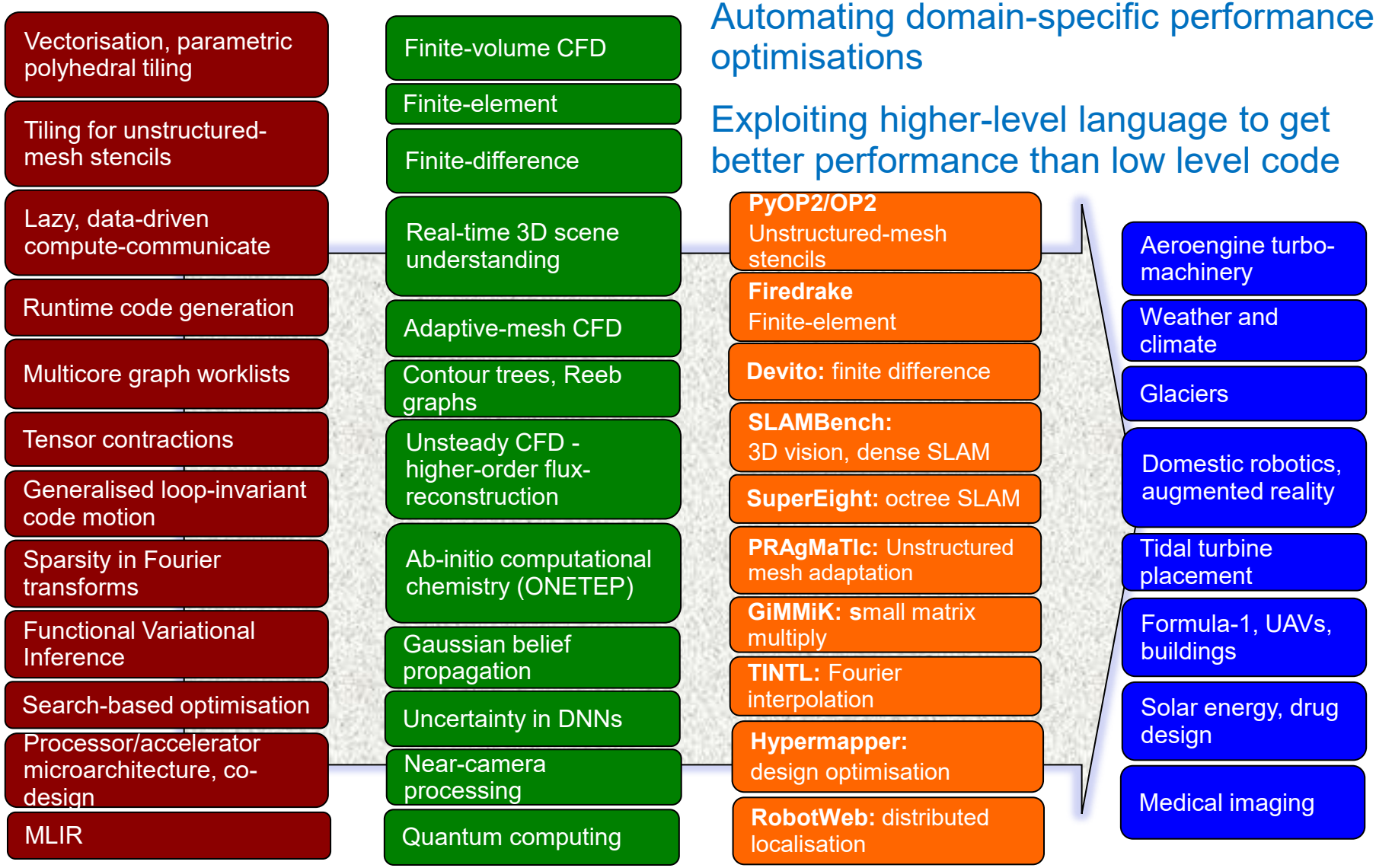
- I aim to show you that you **can** simultaneously
  - **raise the level** at which programmers can reason about code,
  - **provide the compiler** with a model of the computation that enables it to **generate faster code** than you could reasonably write by hand

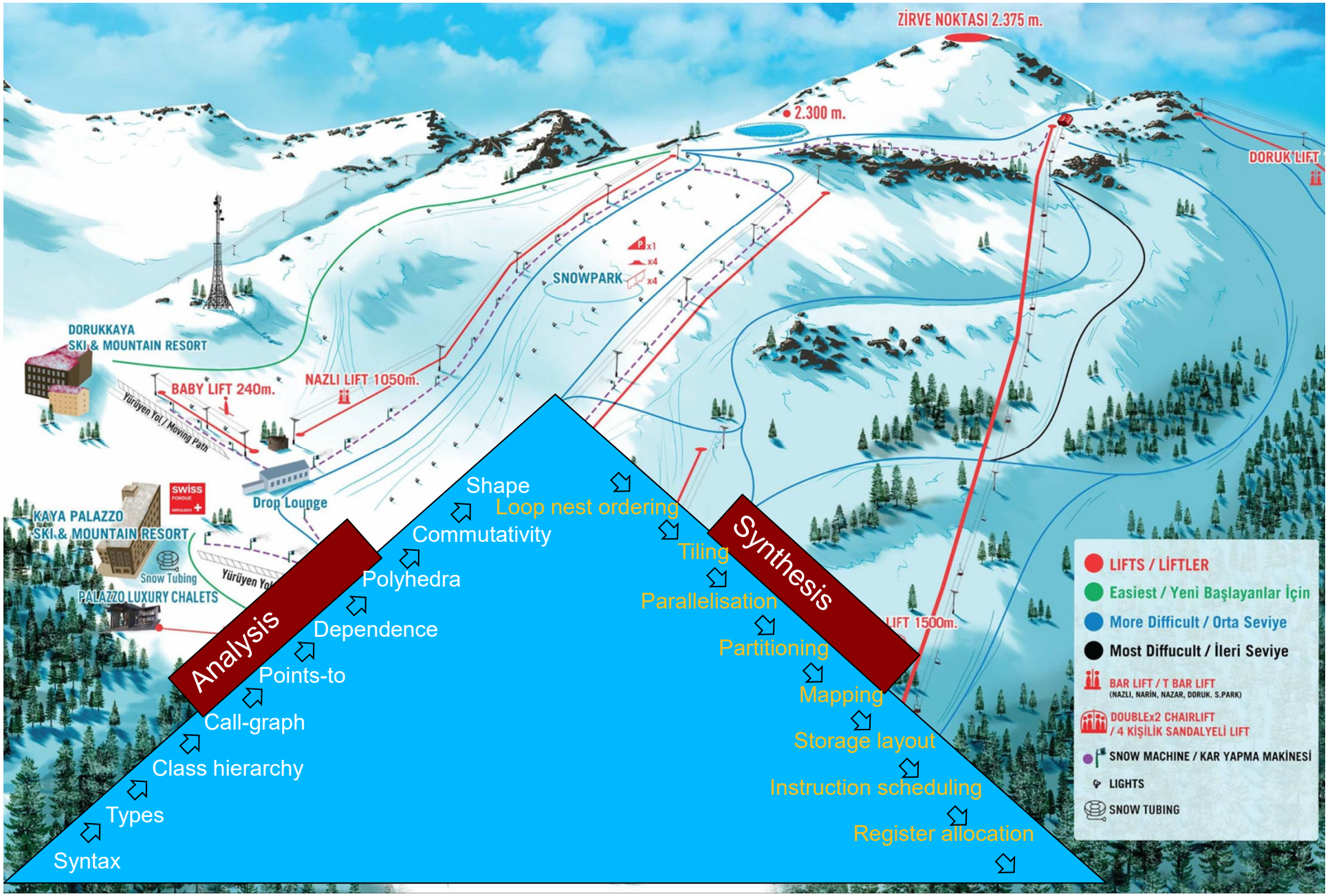
Imperial College London

The work of my research group

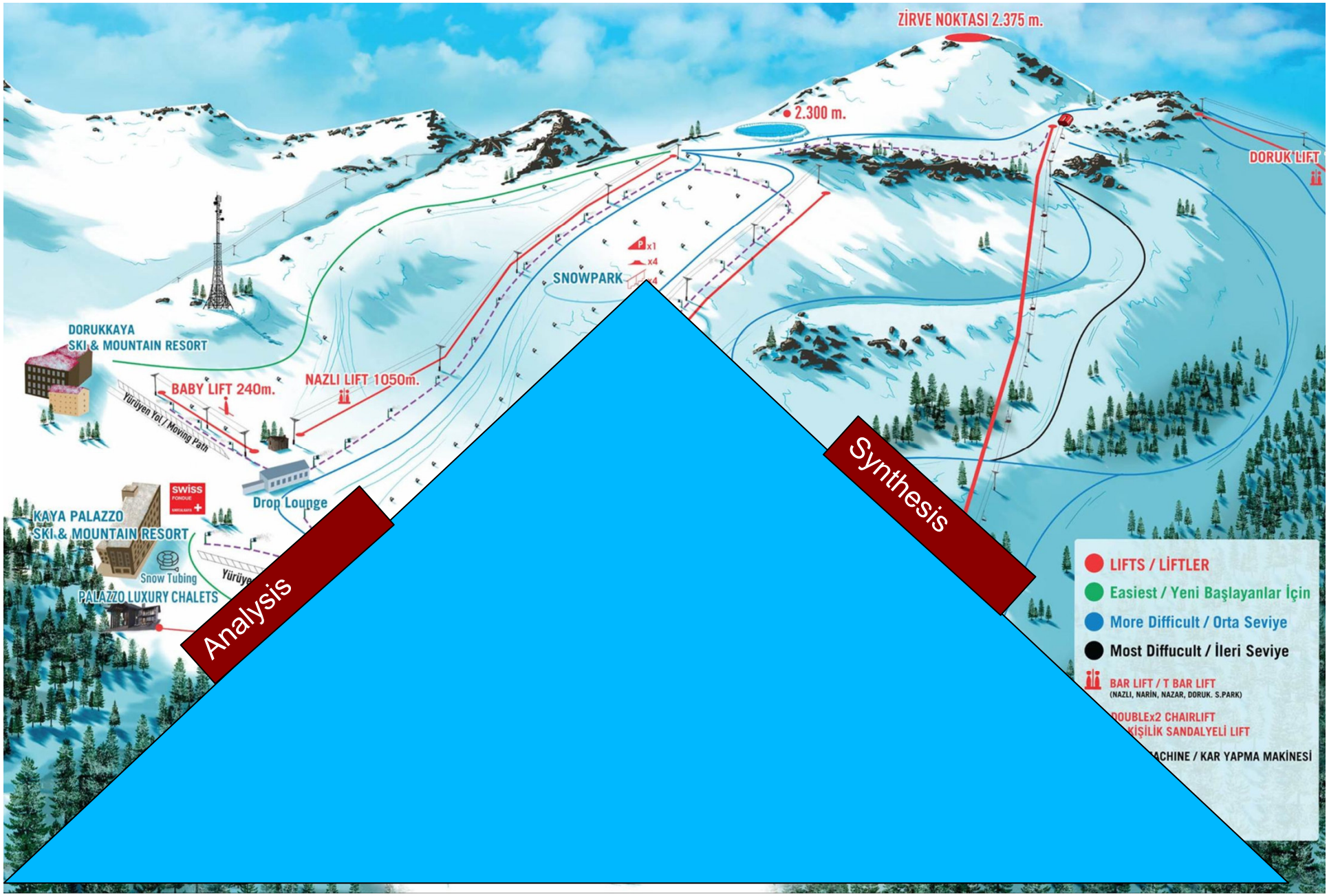
Automating domain-specific performance optimisations

Exploiting higher-level language to get better performance than low level code

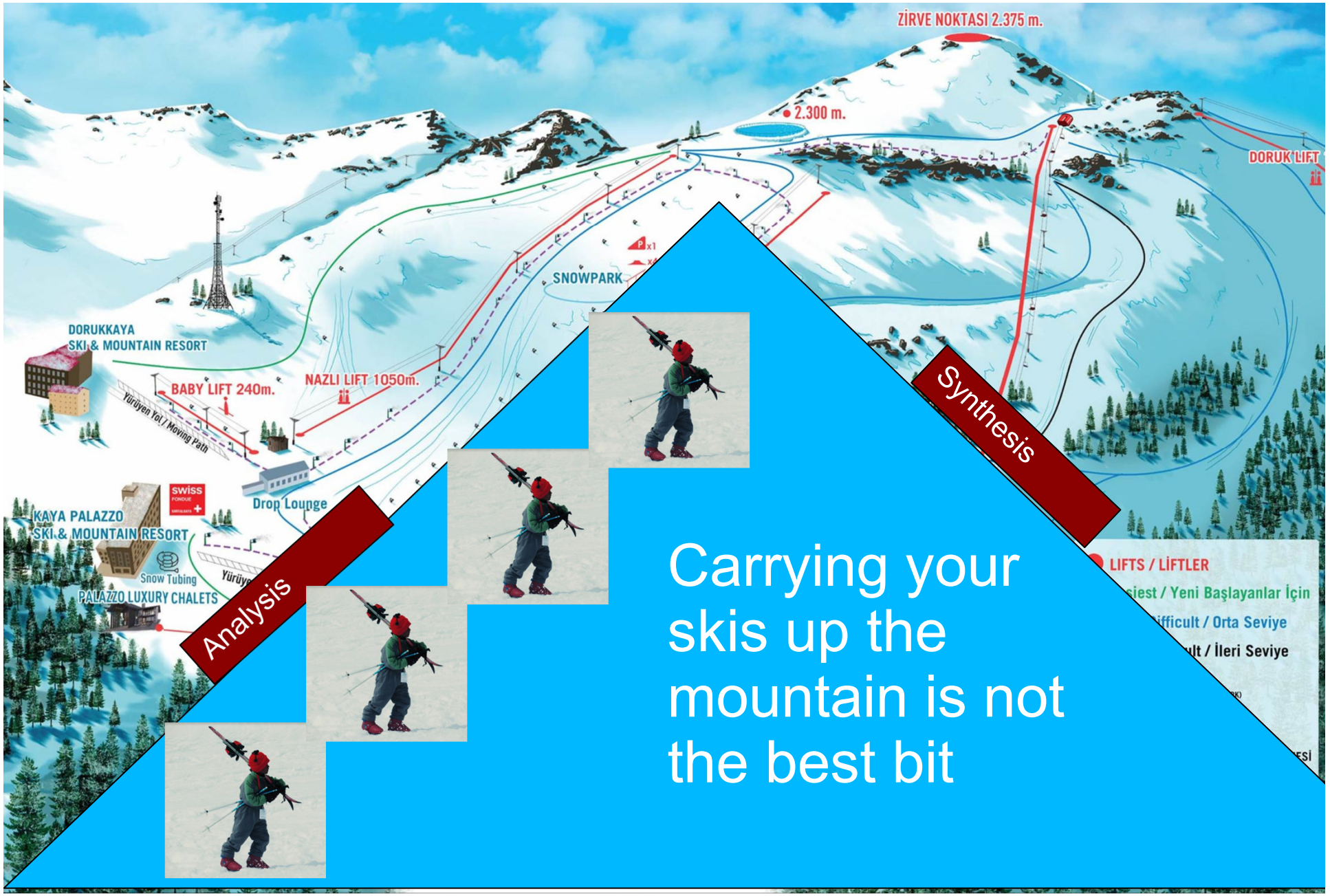




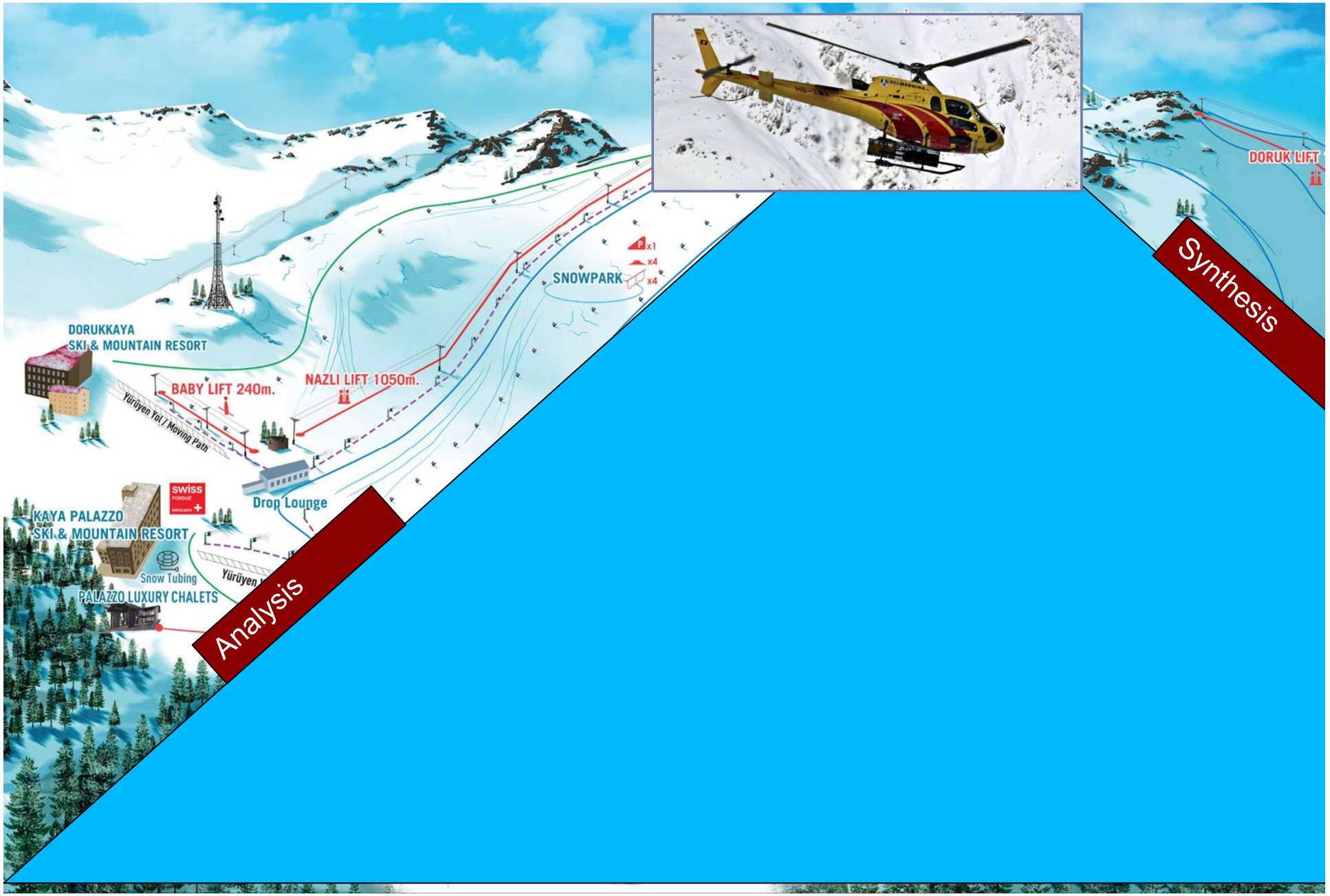
Compilation is like skiing



Compilation is like skiing



Compilation is like skiing



Compilation is like skiing

Plan

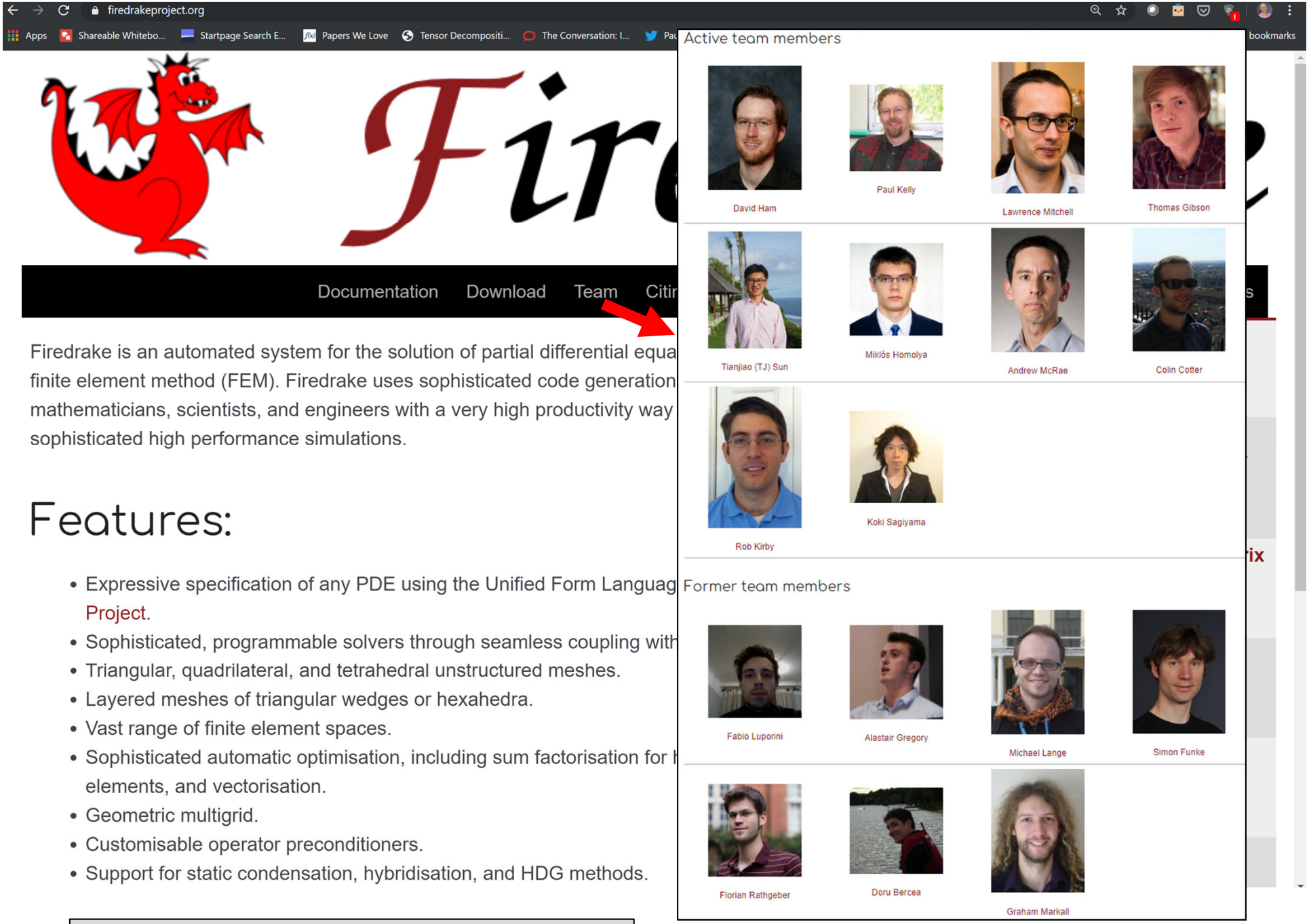
- Compiling is like skiing
  - Analysis is an uphill struggle
- "Turing Tax"
  - The price you pay for running on a general-purpose computer rather than a specialised one
- What do we call…
  - ***The price you pay for using a general-purpose programming language rather than a DSL?***
- This talk:
  - DSLs really can deliver – my examples: Firedrake, Devito
  - DSL compiler architecture: how do DSLs win?
  - Making the DSL ecosystem work

(For more on the Turing Tax see The von Neumann Bottleneck and the Turing Tax - YouTube)

Compilation is like skiing

Example DSL:

- What *is* Firedrake?

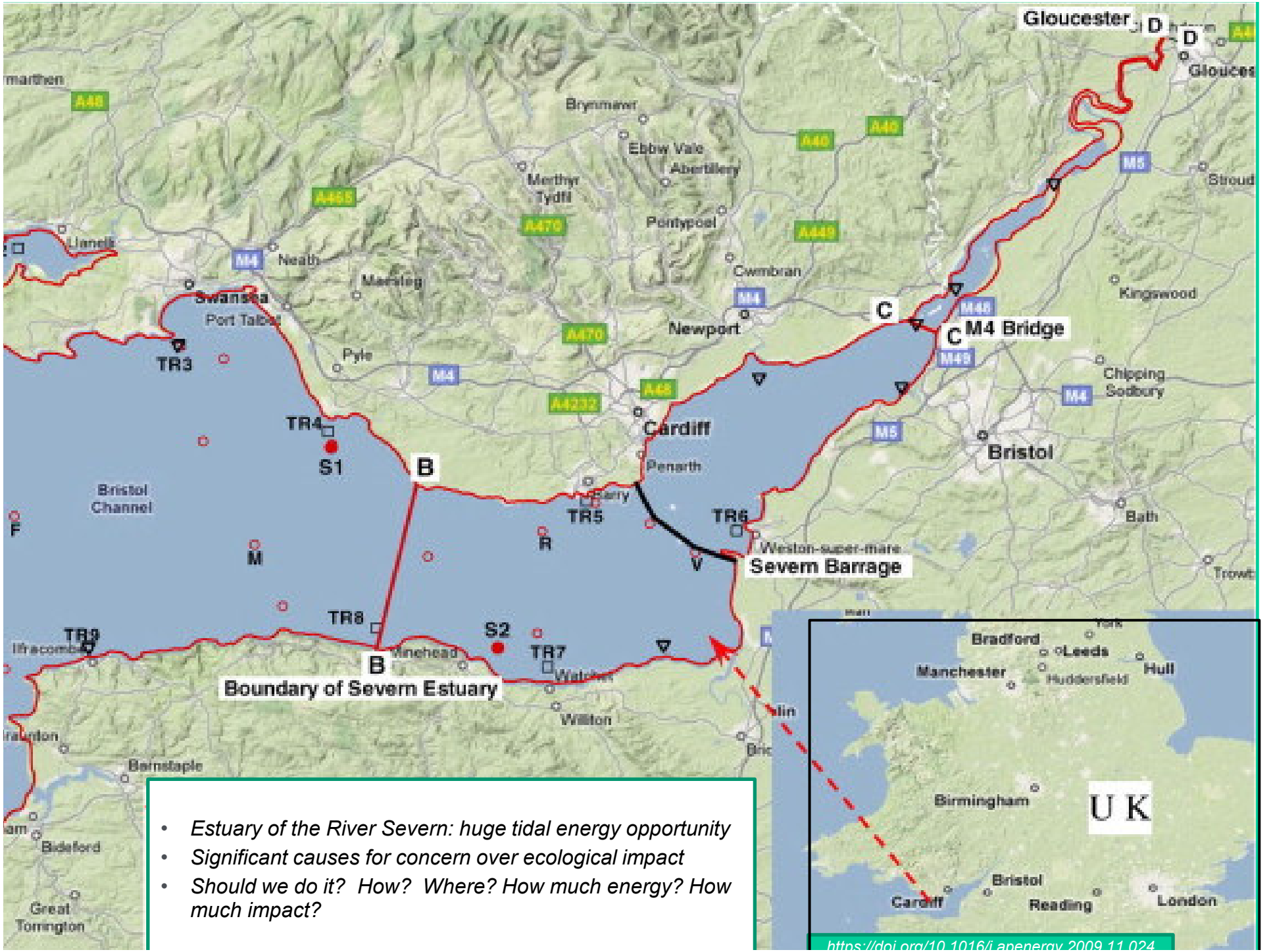


What *is* Firedrake?

Imperial College London

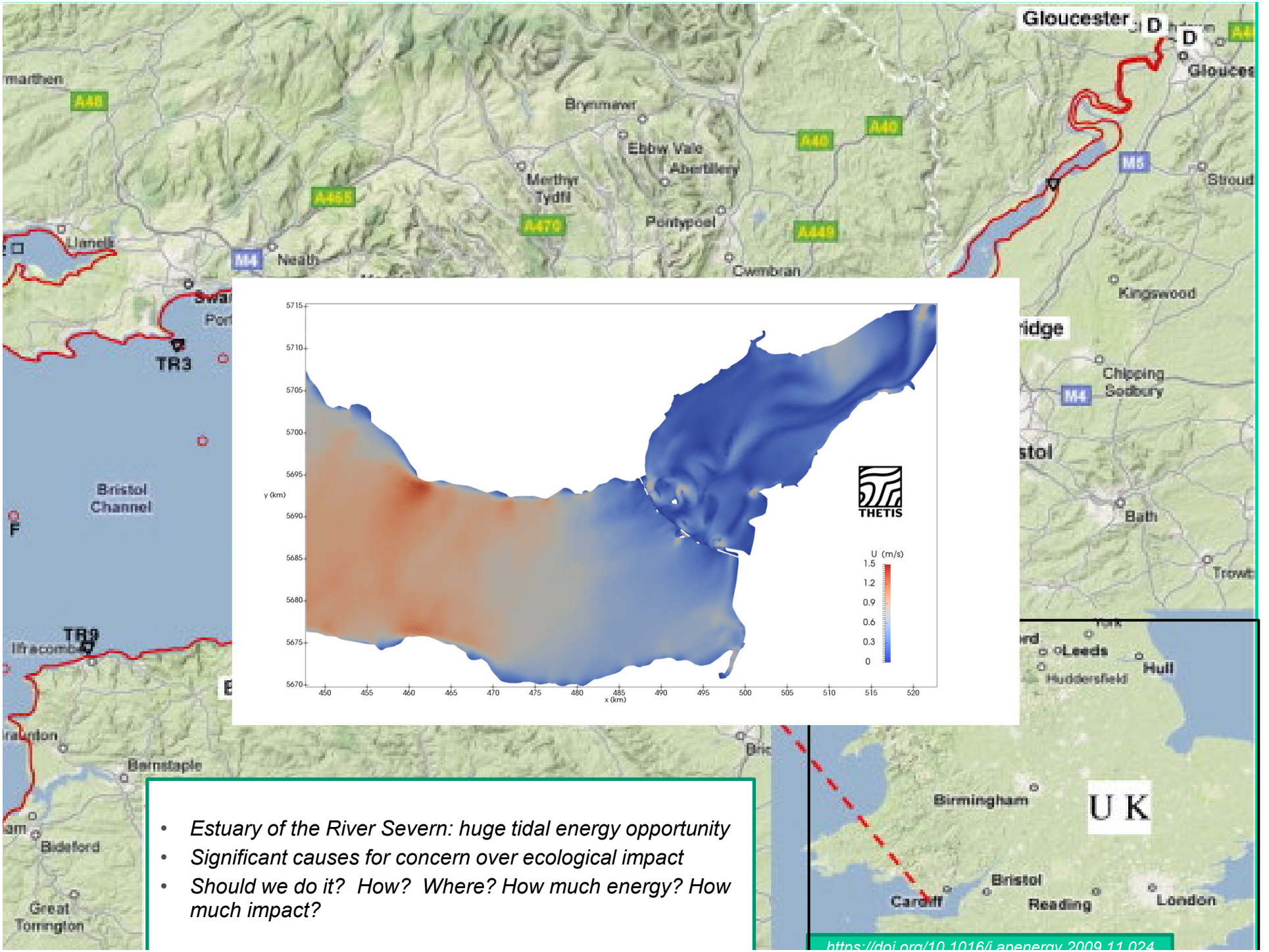
- Firedrake is used in:
  - **Thetis**: unstructured grid coastal modelling framework

- What is it used for? By whom?



- *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?*

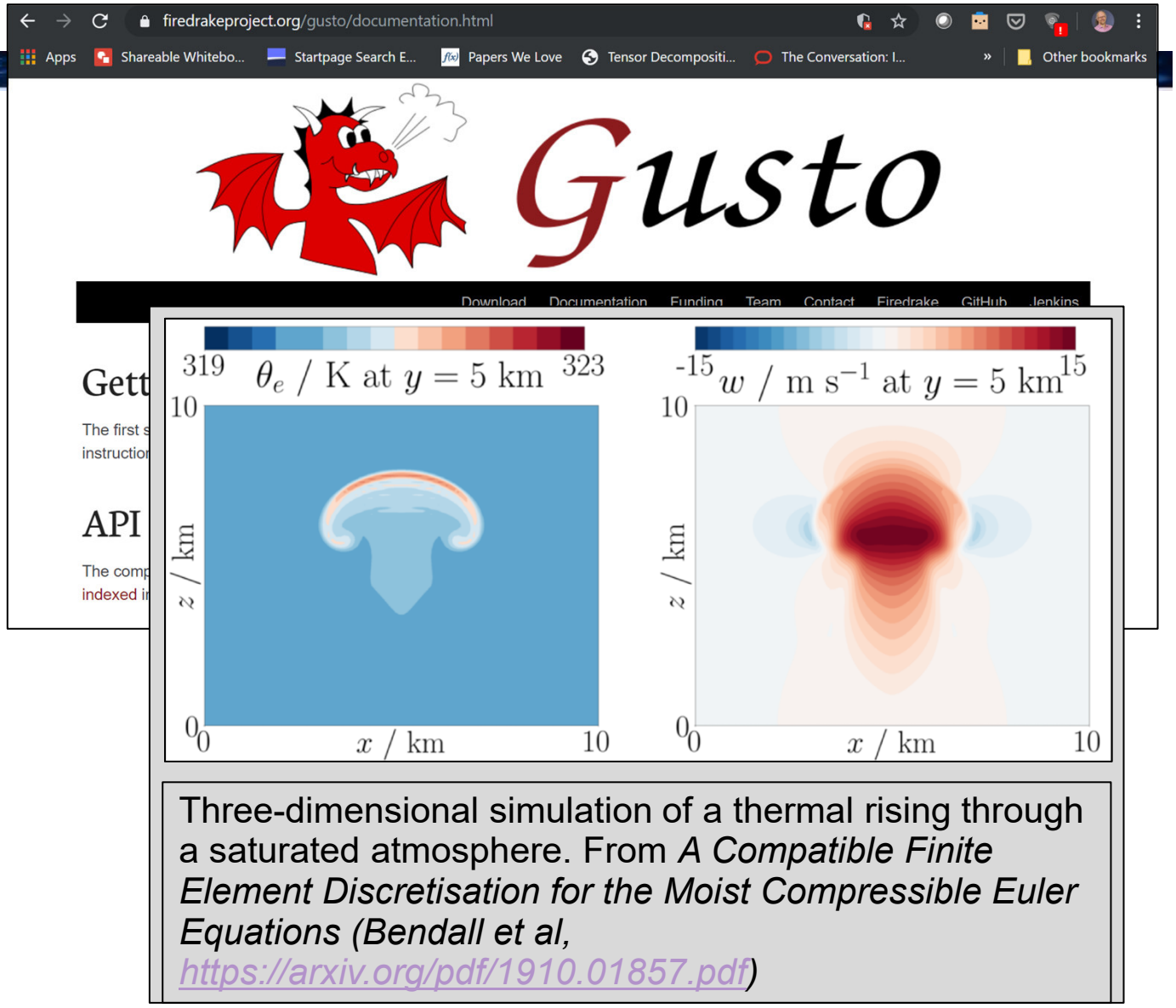
https://doi.org/10.1016/j.apenergy.2009.11.024



- *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?*

https://doi.org/10.1016/j.apenergy.2009.11.024

- Firedrake is used in:
  - **Gusto**: atmospheric modelling framework being used to prototype the next generation of weather and climate simulations for the UK Met Office

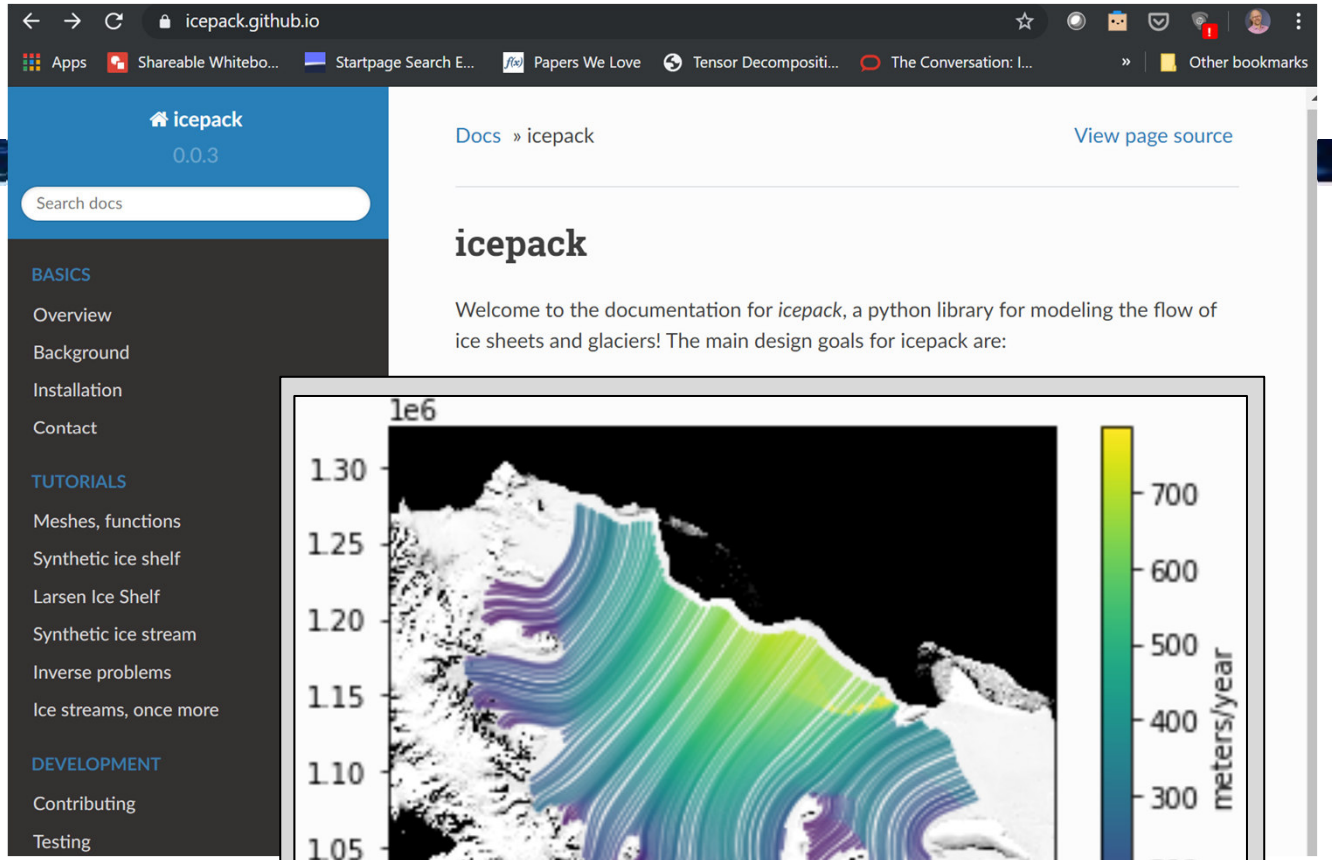


Three-dimensional simulation of a thermal rising through a saturated atmosphere. From *A Compatible Finite Element Discretisation for the Moist Compressible Euler Equations (Bendall et al, https://arxiv.org/pdf/1910.01857.pdf)*

- What is it used for? By whom?

Imperial College London

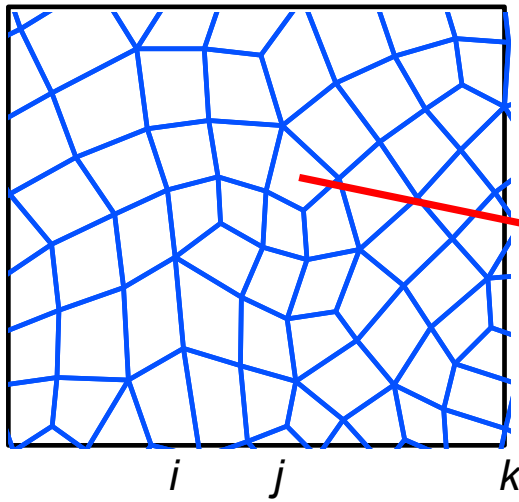
- Firedrake is used in:
  - **Icepack**: a framework for modeling the flow of glaciers and ice sheets, developed at the Polar Science Center at the University of Washington

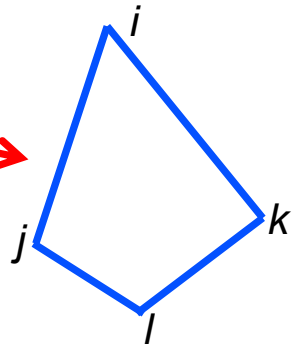


Larsen ice shelf model, from the Icepack tutorial by Daniel Shapero (*https://icepack.github.io/icepack.demo.02-larsen-ice-shelf.html*)

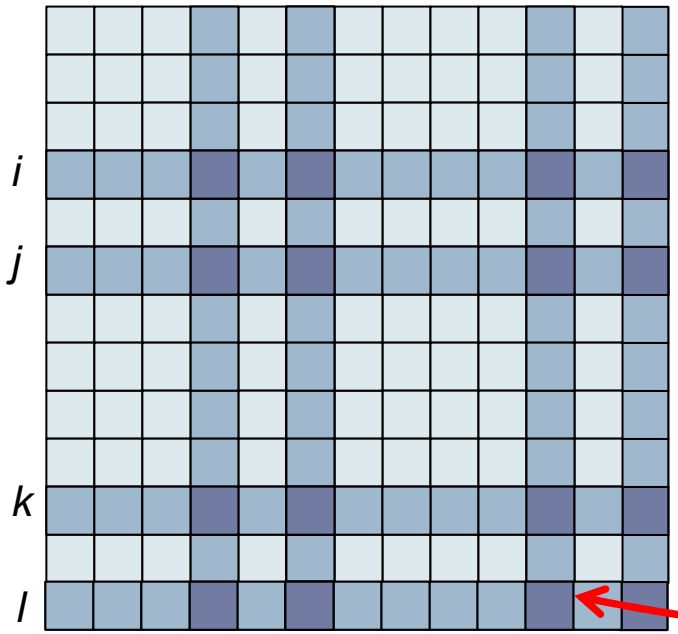
- What is it used for? By whom?

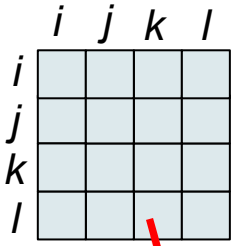
The finite element method in outline



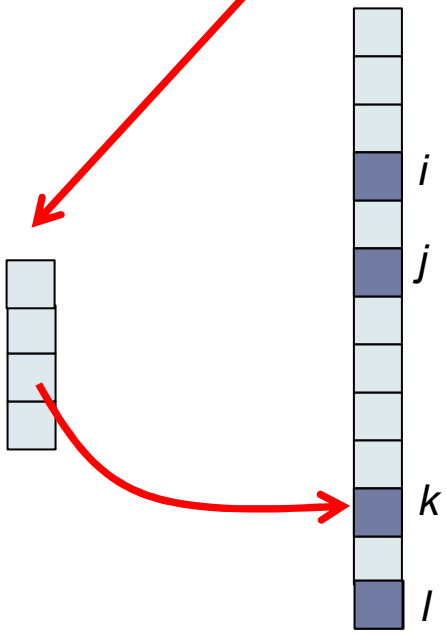



```
do element = 1,N
   assemble(element):
   ∫_Ω vL(u^δ)dX = ∫_Ω vq dX.
end do
```





$$Ax = b$$



- Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector

Imperial College London

Multilayered abstractions for FE

- Local assembly:
  - Computes local assembly matrix
  - Using:
    - The (weak form of the) PDE
    - The discretisation
  - Key operation is evaluation of expressions over basis function representation of the element
- Mesh traversal:
  - *PyOP2*
  - *Loops over the mesh*
  - *Key is orchestration of data movement*
- Solver:
  - Interfaces to standard solvers through PetSc

Imperial College London

Example: Burgers equation

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

- 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)

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"))
```

- Firedrake implements the Unified Form Language (UFL)
- Embedded in Python

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

- 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
```

- (UFL is also the DSL of the FEniCS project)
- What does its DSL actually look like?

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

# We choose degree 2 continuous Lagrange polynomials.
# 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

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
# initial condition as our starting gue

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

# :math:`\nu` is set to a (fairly arbit

nu = 0.000

timestep =

# Define t

F = (inner
     + inn

outfile =

outfile.wr

# Finally,

t = 0.0
end = 0.5
while (t <
    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"))
```

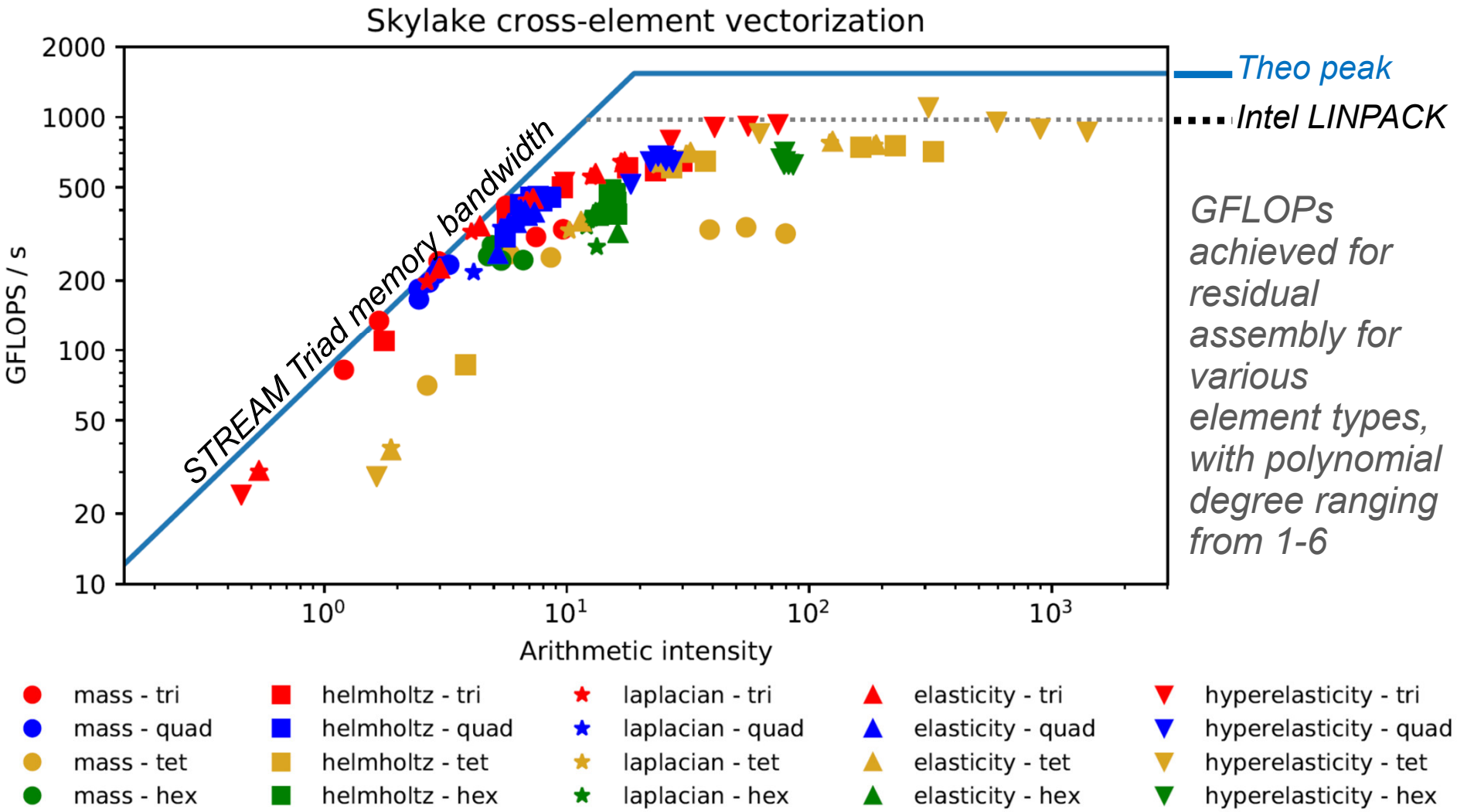
- What does its DSL actually look like?

```
#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_t45 + form_t18[6 * form_ip + form_j0] * form_t38[form_k0_0] + form_t44;
          t2[1 + 24 * form_j0 + 2 * form_k0_0] = t2[1 + 24 * form_j0 + 2 * form_k0_0] + form_t18[6 * form_ip + form_k0_0] * 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);
  }
}
```

- 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

Firedrake: single-node AVX512 performance

- Does it generate good code?



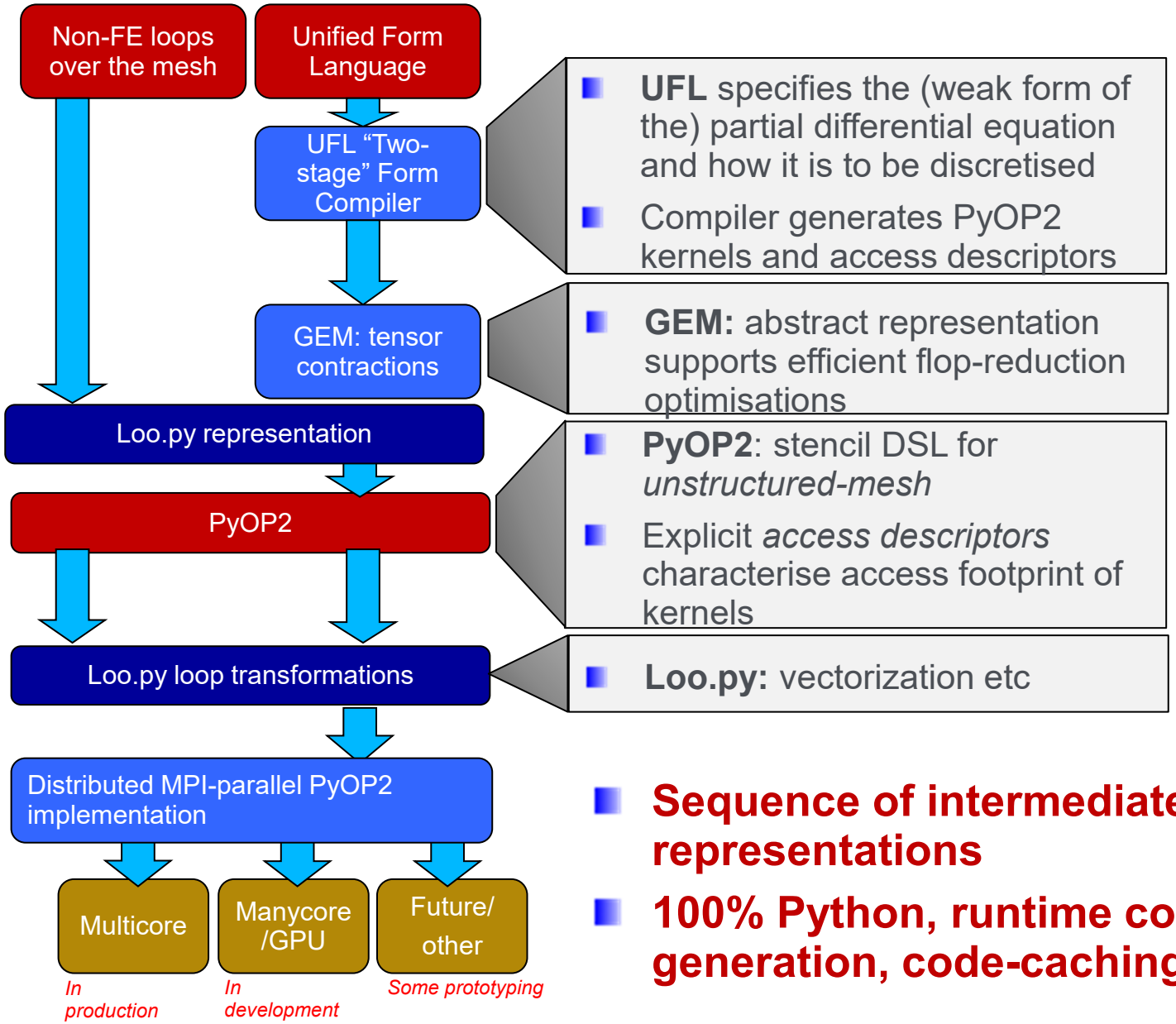
GFLOPs achieved for residual assembly for various element types, with polynomial degree ranging from 1-6

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

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

Imperial College London

Firedrake: compiler architecture



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

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

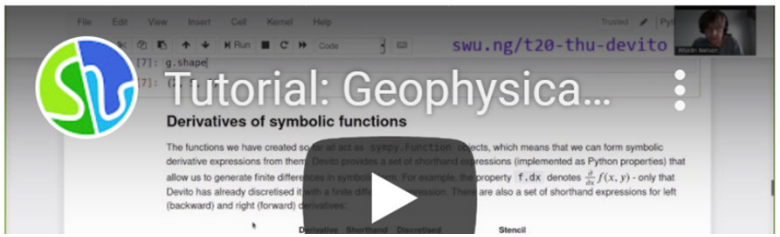
Another example DSL:

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!

Imperial College London

Devito: applications

- 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

Imperial College London

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/mo
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:

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

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

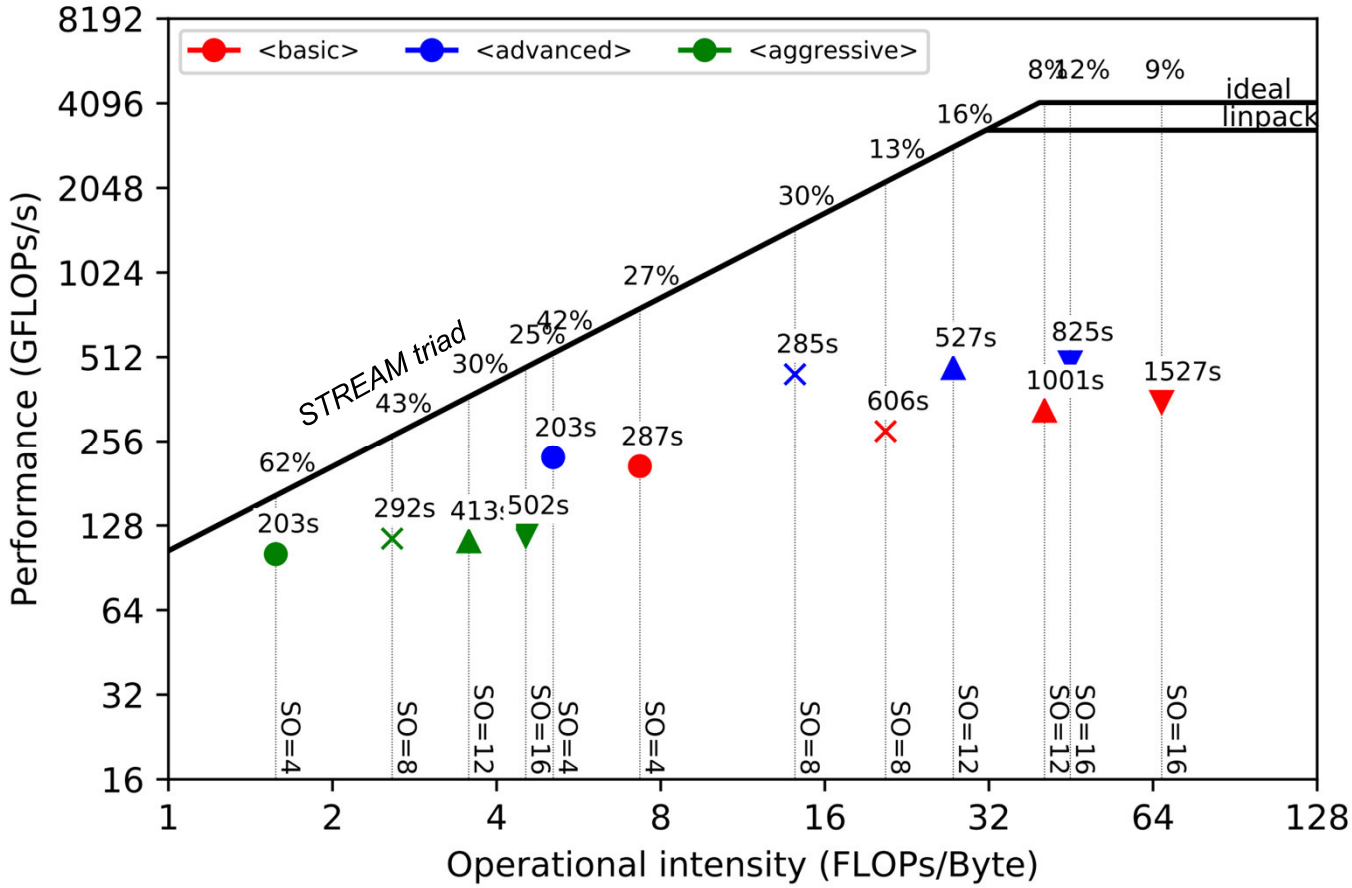
We derive and generate the stencil operator code, then run it a specified number of timesteps

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

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

Imperial College London

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.

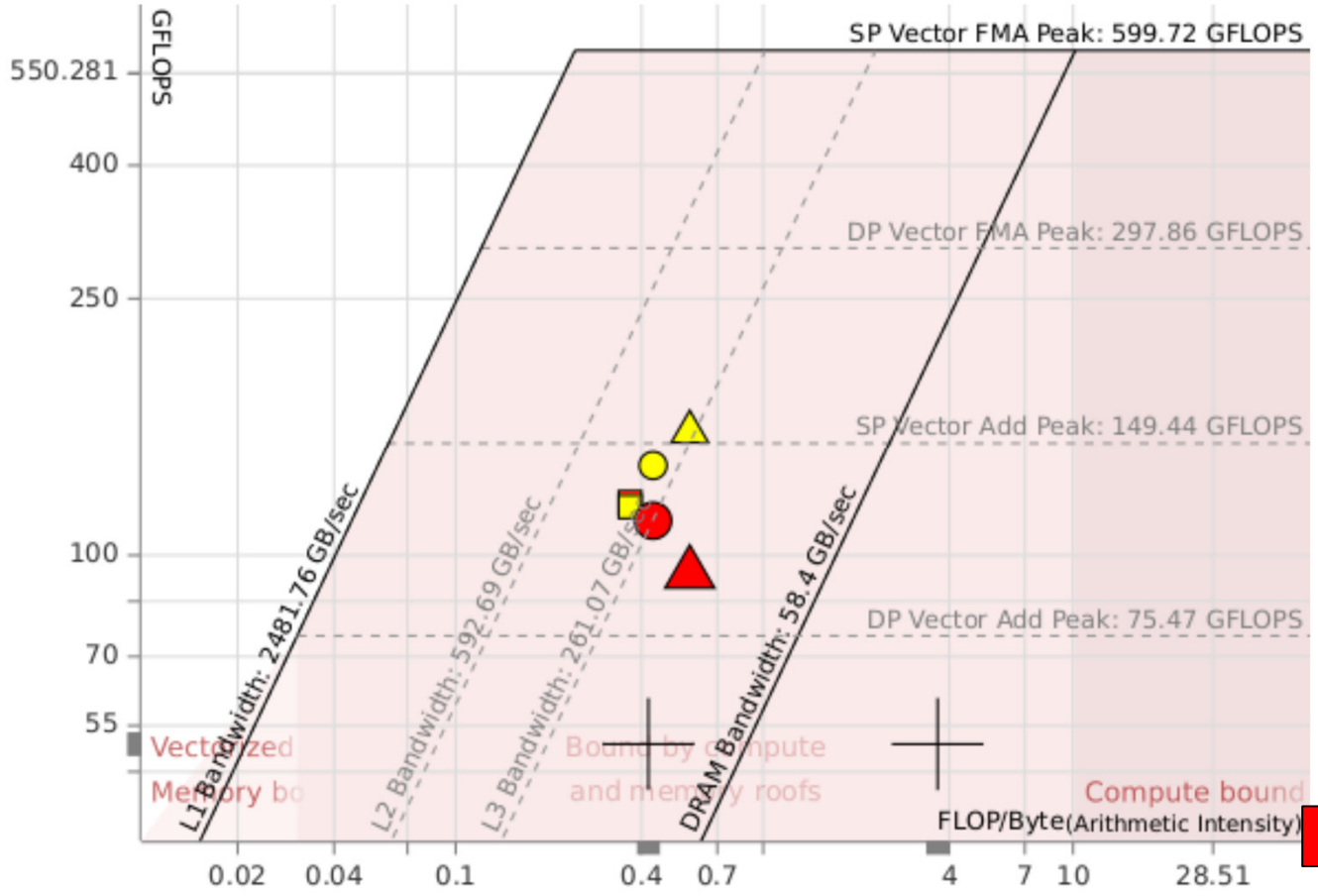
Space order:
4 (circles),
8 (crosses), and
12 (triangles)
16 (nablas)

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

For latest performance data see https://www.devitoproject.org/thematrix/

Imperial College London

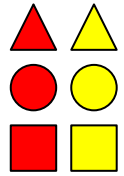
Devito: tiling-in-time



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 **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 iarXiv:2010.10248

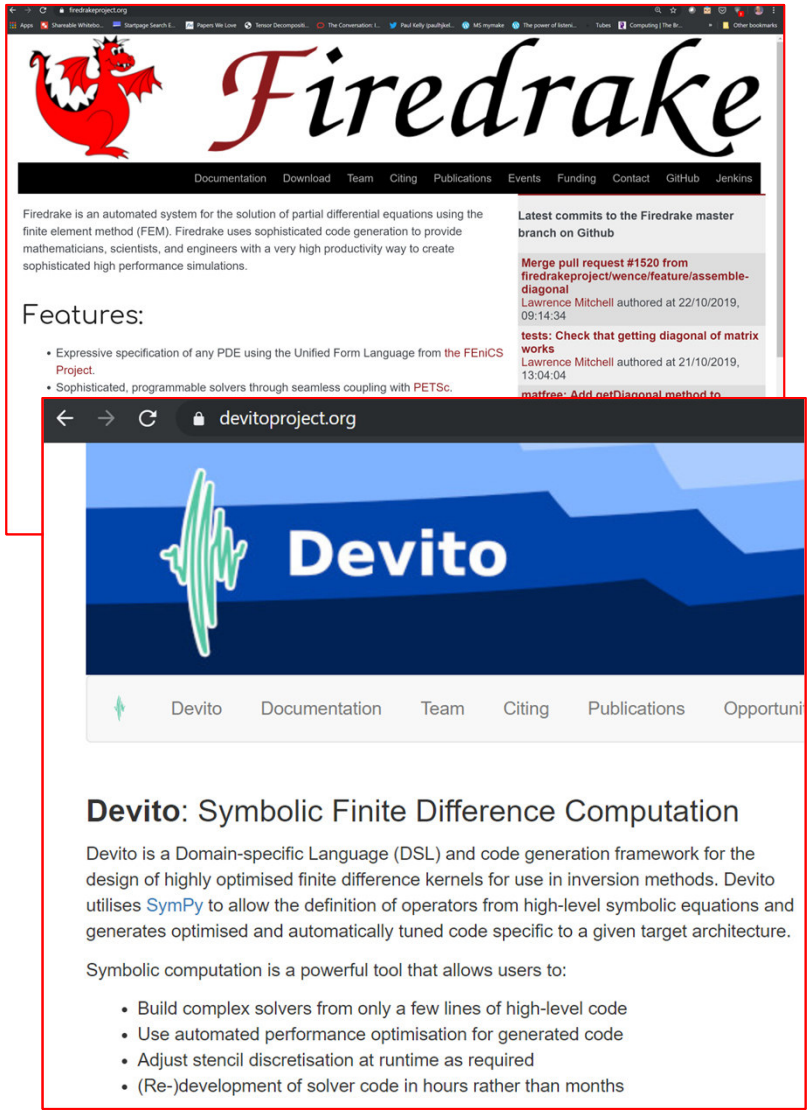
Imperial College London

Why I do what I do, and what I've learned

- **Engaging with applications to exploit domain-specific optimisations can be incredibly fruitful**
  - Compiling general purpose languages is worthy but usually incremental
- **Compiler architecture is all about designing intermediate representations – that make hard things look easy**
  - Tools to deliver domain-specific optimisations often have domain-specific representations
  - Premature lowering is the constant enemy (appropriate lowering is great)
- **Along the way, we learn something about building better general-purpose compilers and programming abstractions**
  - Drill vertically, expand horizontally

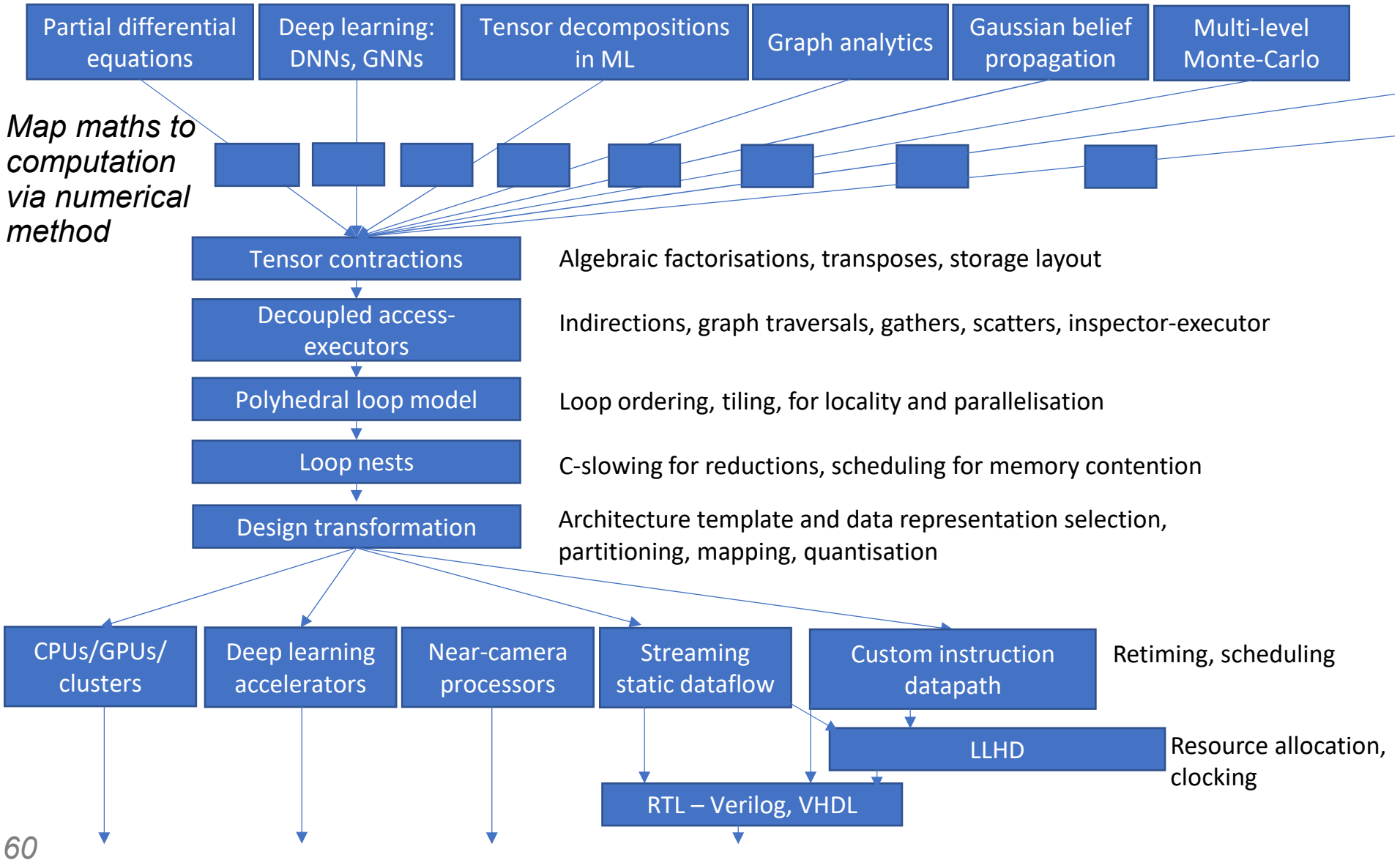
How can we change the world?

- **The real value of Firedrake and Devito is in supporting the applications users in exploring *their* design space**
- **We enable them to navigate rapidly through alternative solutions to their problem**
- **In the future, we will have automated pathways from maths to code for many classes of problem, and many alternative solution techniques**



Imperial College London

Vision for the future



Imperial College London

Acknowledgements

Thank you to our many many collaborators!

Partly funded/supported by

- SysGenX: Composable software generation for system-level simulation at Exascale (EP/W026066/1)
- XDSL: Efficient Cross-Domain DSL Development for Exascale (EP/W007789/1)
- NERC Doctoral Training Grant (NE/G523512/1)
- EPSRC “MAPDES” project (EP/I00677X/1)
- EPSRC “PSL” project (EP/I006761/1)
- Rolls Royce and the TSB through the SILOET programme
- EPSRC “PAMELA” Programme Grant (EP/K008730/1)
- EPSRC “PRISM” Platform Grants (EP/I006761/1 and EP/R029423/1)
- EPSRC “Custom Computing” Platform Grant (EP/I012036/1)
- EPSRC “Application Customisation” Platform Grant (EP/P010040/1)
- EPSRC “A new simulation and optimisation platform for marine technology” (EP/M011054/1)
- Basque Centre for Applied Mathematics (BCAM)

- Code:
  - http://www.firedrakeproject.org/
  - http://op2.github.io/PyOP2/
  - https://github.com/OP-DSL/OP2-Common

Imperial College London

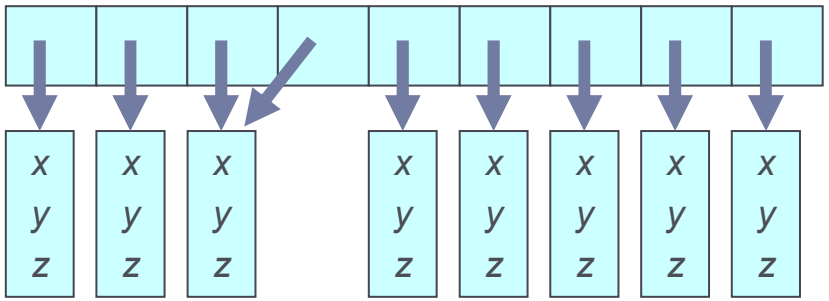
Extra slides for questions

Easy parallelism

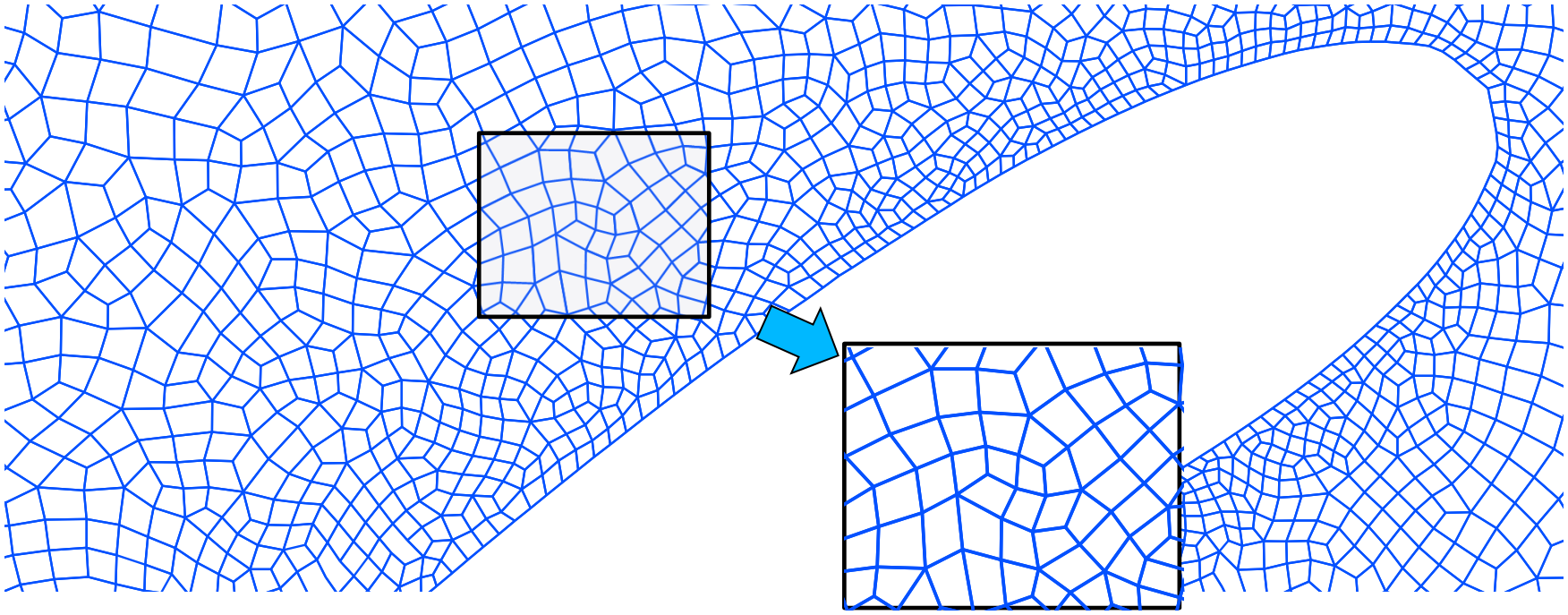
Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- Can the iterations of this loop be executed in parallel?



- Oh no: not all the iterations are independent!
  - You want to re-use piece of code in different contexts
  - Whether it's parallel depends on context!



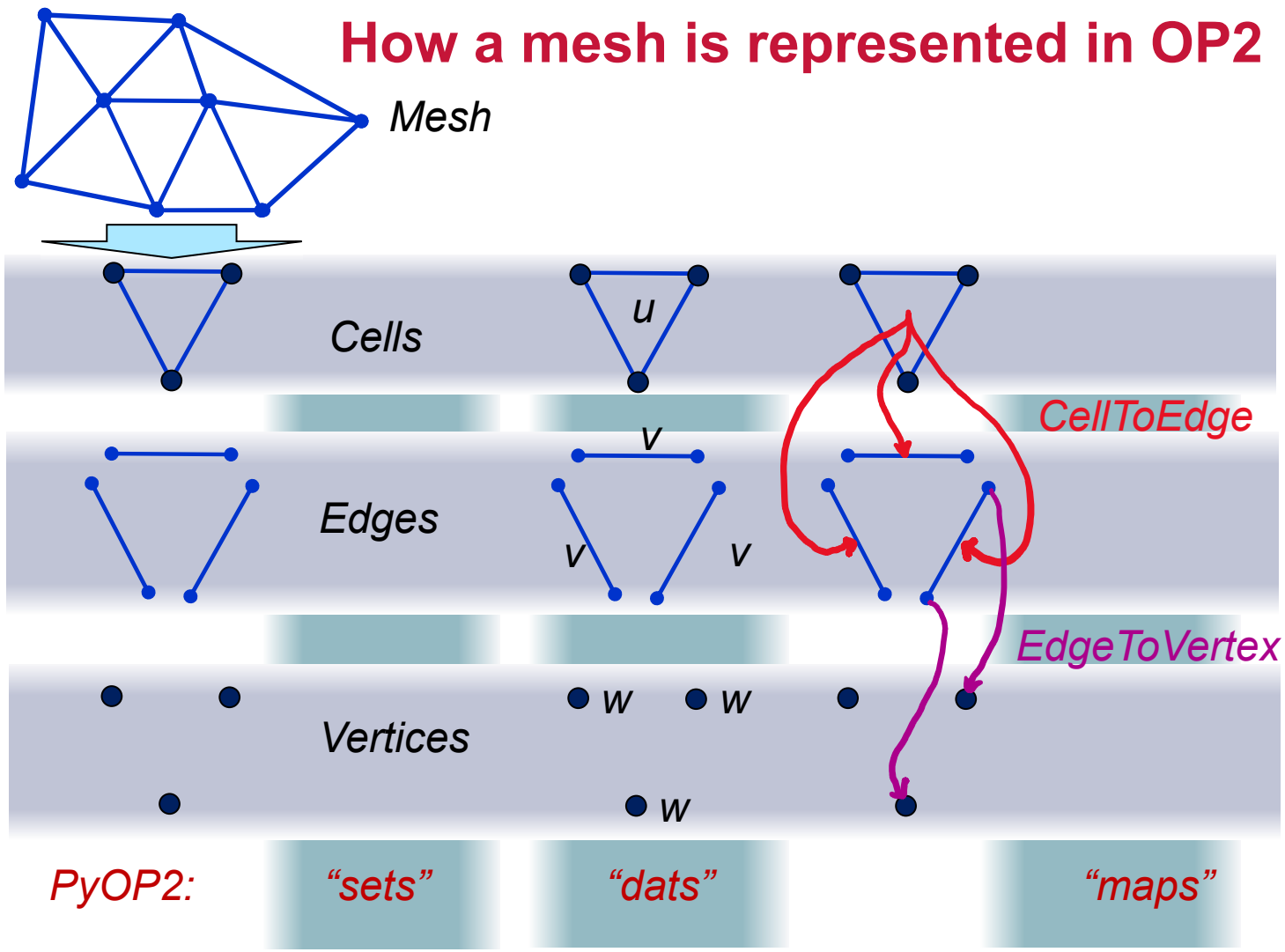
- Unstructured meshes require pointers/indirection because adjacency lists have to be represented explicitly
- A controlled form of pointers (actually a general graph)

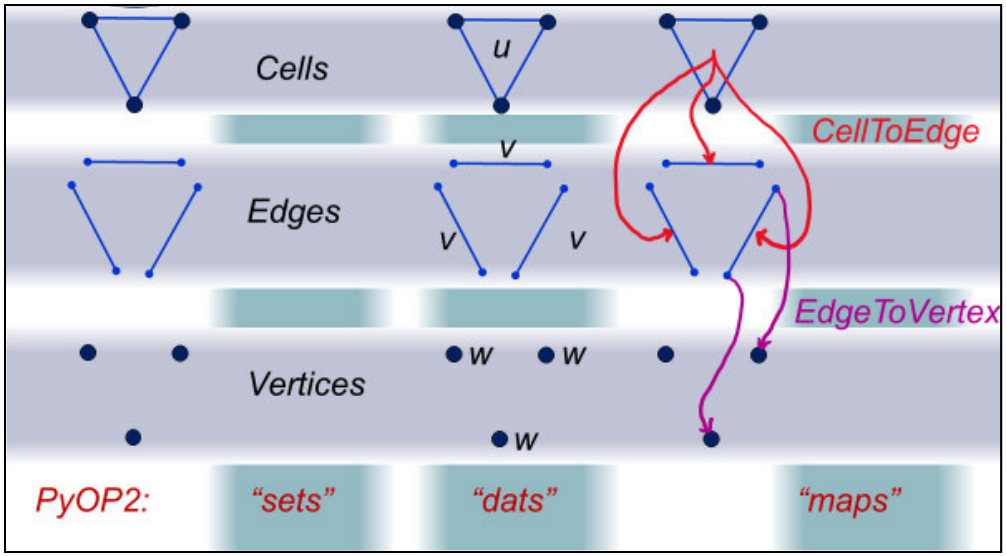
- **OP2** is a C++ and Fortran library for parallel loops over the mesh, implemented by source-to-source transformation
- **PyOP2** is the same basic model, implemented in Python using runtime code generation

- Enables generation of highly-optimised vectorised, CUDA, OpenMP and MPI code

- The OP2 model originates from Oxford (Mike Giles et al)

How a mesh is represented in OP2





OP2 loops, access descriptors and kernels

op_par_loop(set, kernel, access descriptors)

We specify which **set** to iterate over

We specify a **kernel** to execute – the kernel operates entirely locally, on the **dats** to which it has access

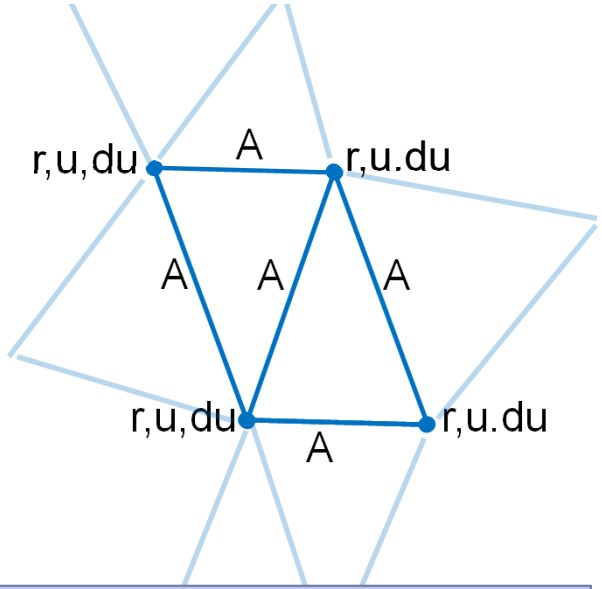
The **access descriptors** specify which dats the kernel has access to:

- **Which dats of the target set**
- **Which dats of sets indexed from this set through specified maps**

- OP2 separates local (kernel) from global (mesh)
- OP2 makes data dependence explicit

PyOP2: “decoupled access-execute”

- Parallel loops, over sets (nodes, edges etc)
- Access descriptors specify how to pass data to and from the C kernel
- The kernel operates only on local data



Access descriptors specify how to feed the kernel from the mesh

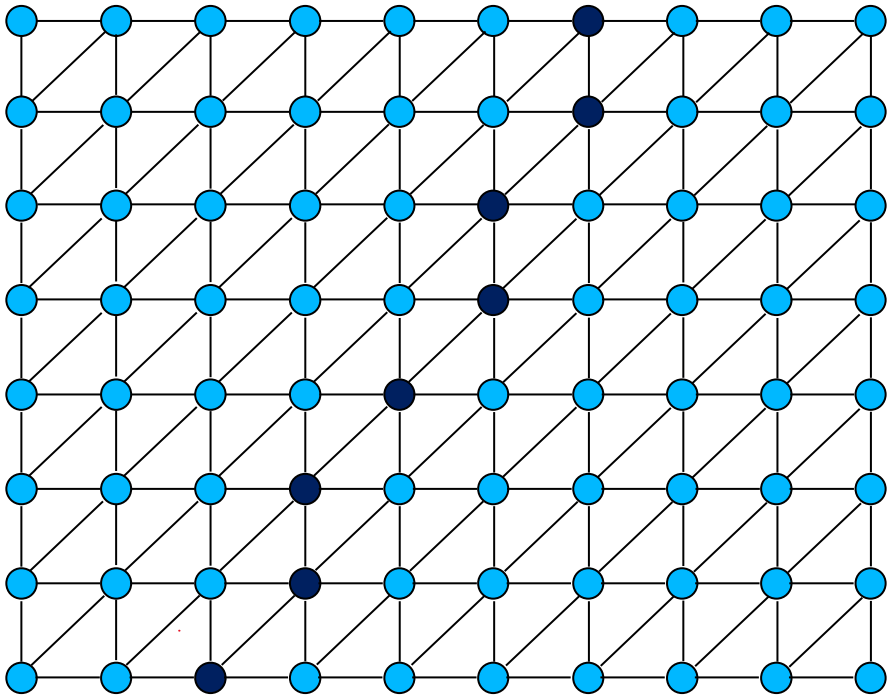
```
for iter in xrange(0, NITER):
    u_sum = op2.Global(1, data=0.0, np.float32)
    u_max = op2.Global(1, data=0.0, np.float32)
    op2.par_loop(res, edges,
        p_A(op2.READ),
        p_u(op2.READ, edge2vertex[1]),
        p_du(op2.INC, edge2vertex[0]),
        beta(op2.READ))
    op2.par_loop(update, nodes,
        p_r(op2.READ),
        p_du(op2.RW),
        p_u(op2.INC),
        u_sum(op2.INC),
        u_max(op2.MAX))
```

```
void res(float *A, float *u, float *du,
         const float *beta) {
  *du += (*beta) * (*A) * (*u);
}
```

```
void update(float *r, float *du, float *u, float
      *u_sum, float *u_max) {
  *u += *du + alpha * (*r);
  *du = 0.0f;
  *u_sum += (*u) * (*u);
  *u_max = *u_max > *u ? *u_max : *u;
}
```

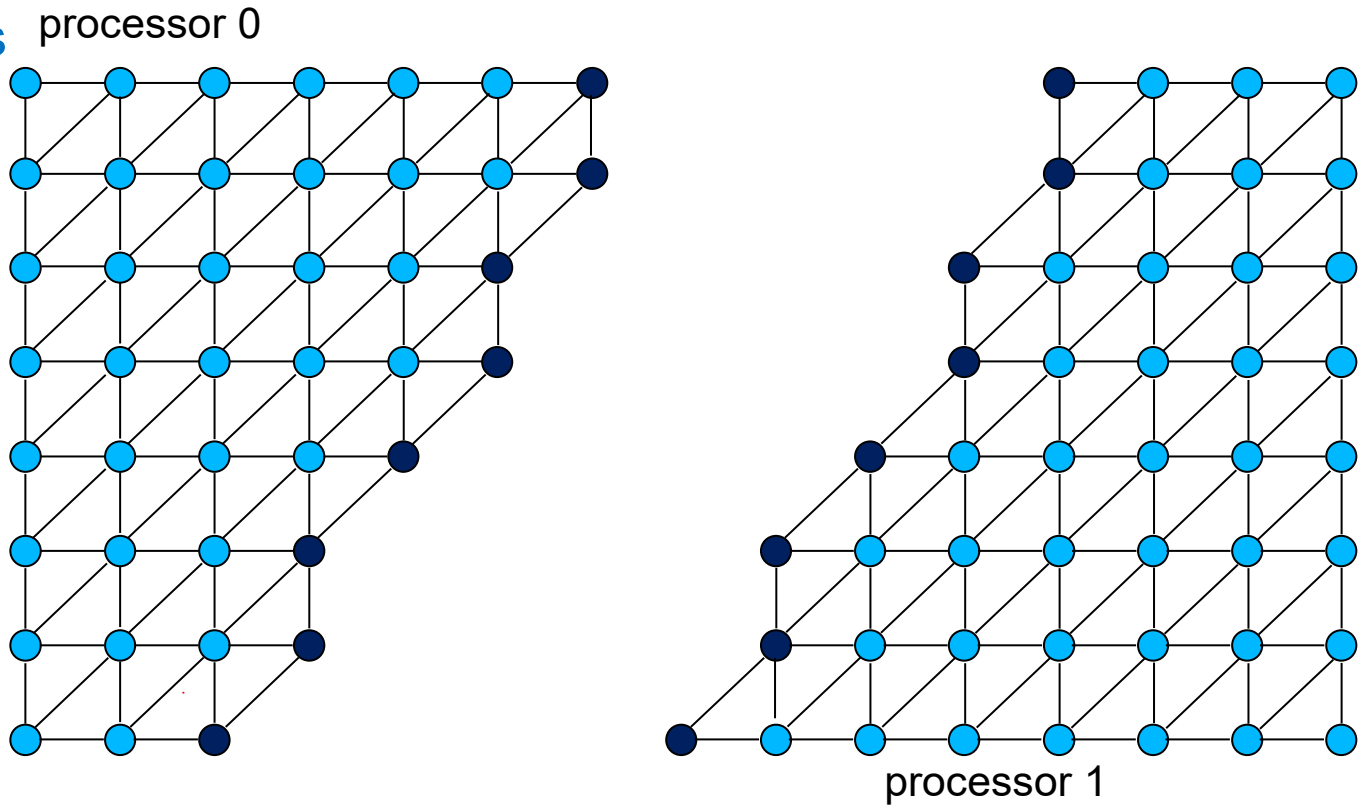
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region



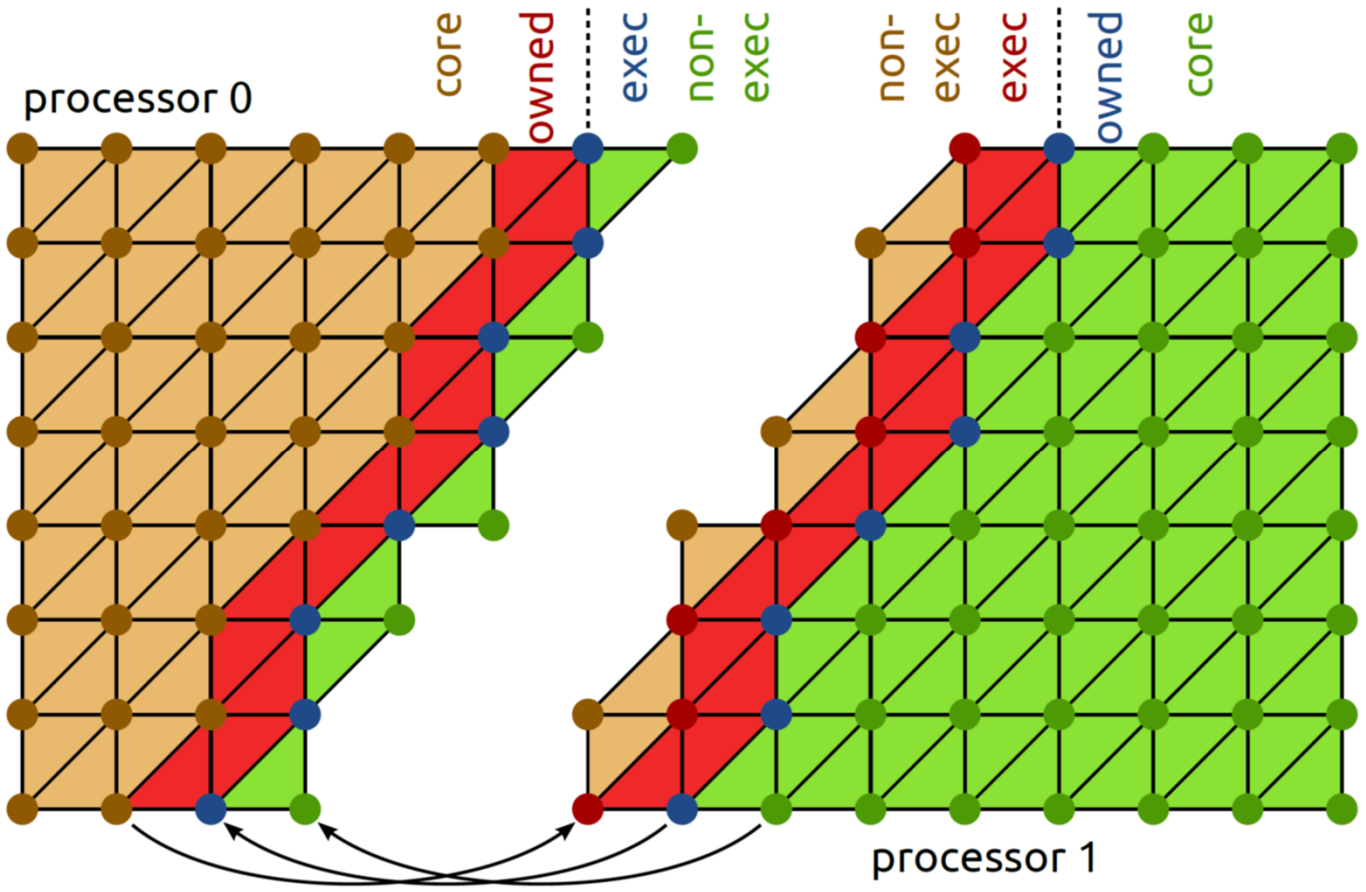
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region



Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PetSC DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region



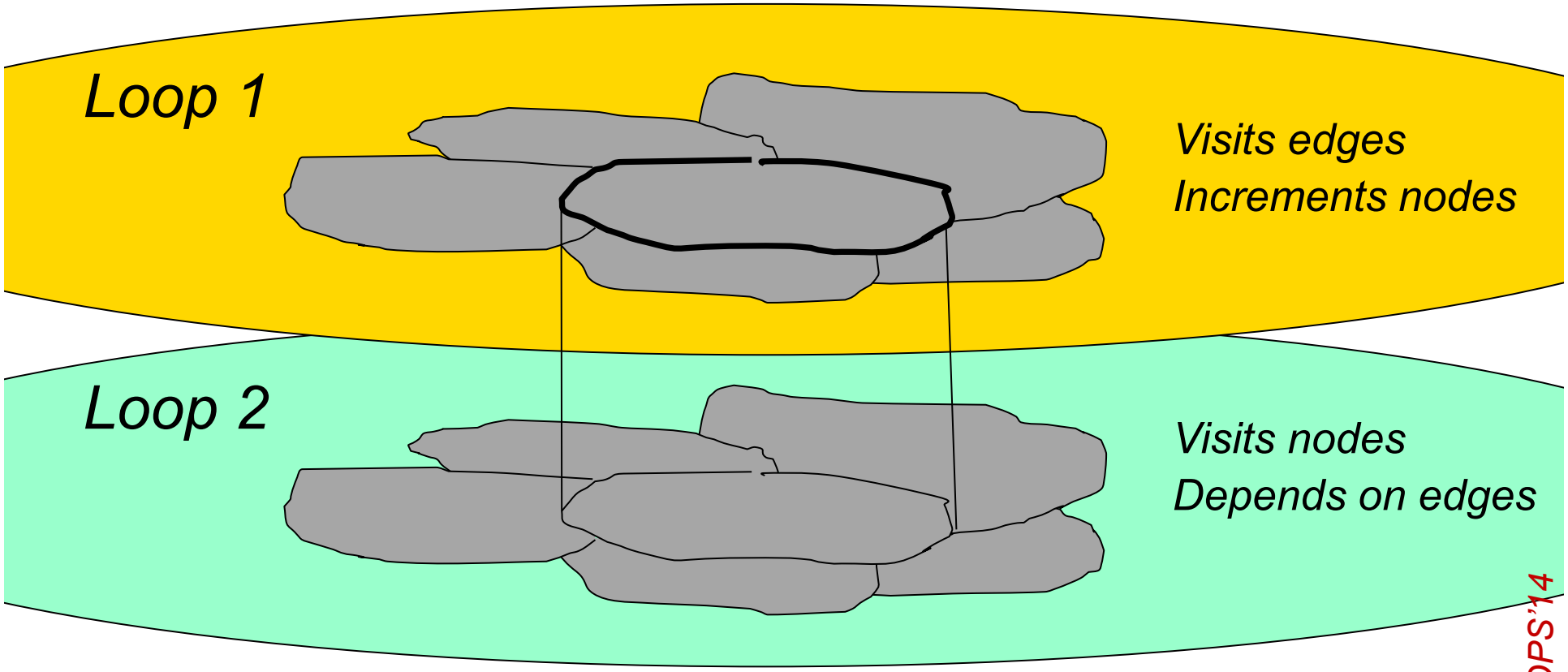
- **Core:** entities owned which can be processed without accessing halo data.
- **Owned:** entities owned which access halo data when processed
- **Exec halo:** off-processor entities which are redundantly executed over because they touch owned entities
- **Non-exec halo:** off-processor entities which are not processed, but read when computing the exec halo

- Can we automate interesting optimisations that would be hard to do by hand?

  - First example:
    - Tiling for cache locality

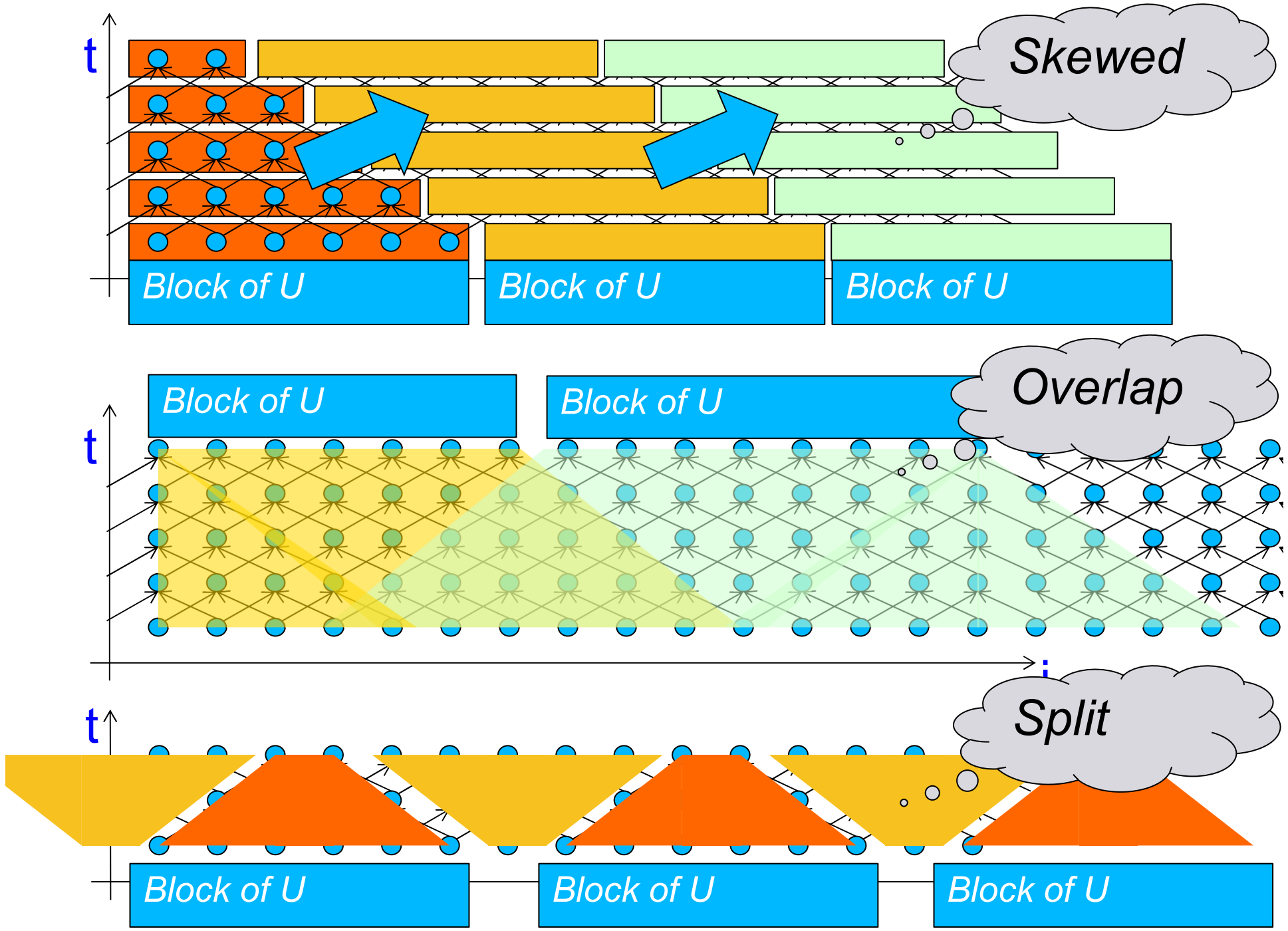
    - (This optimisation has been implemented – and automated – but does not currently form part of the standard distribution)

Sparse split tiling on an unstructured mesh, for locality

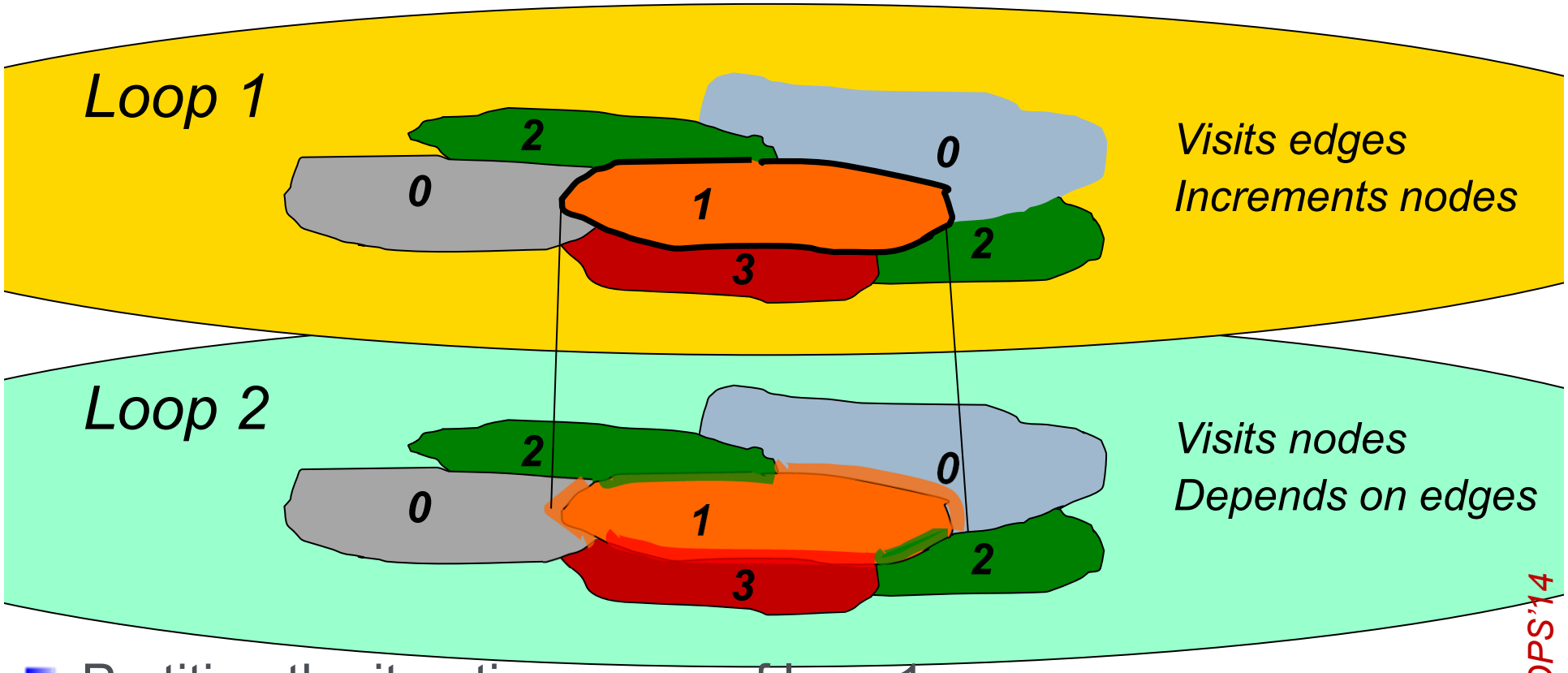


- How can we load a block of mesh and do the iterations of loop 1, then the iterations of loop 2, before moving to the next block?
- If we could, we could dramatically improve the memory access behaviour!

Strout, Luporini et al, IPDPS'14


t
Skewed
Block of U
Block of U
Block of U
Block of U
Block of U
Overlap
t
i
t
Split
Block of U
Block of U
Block of U

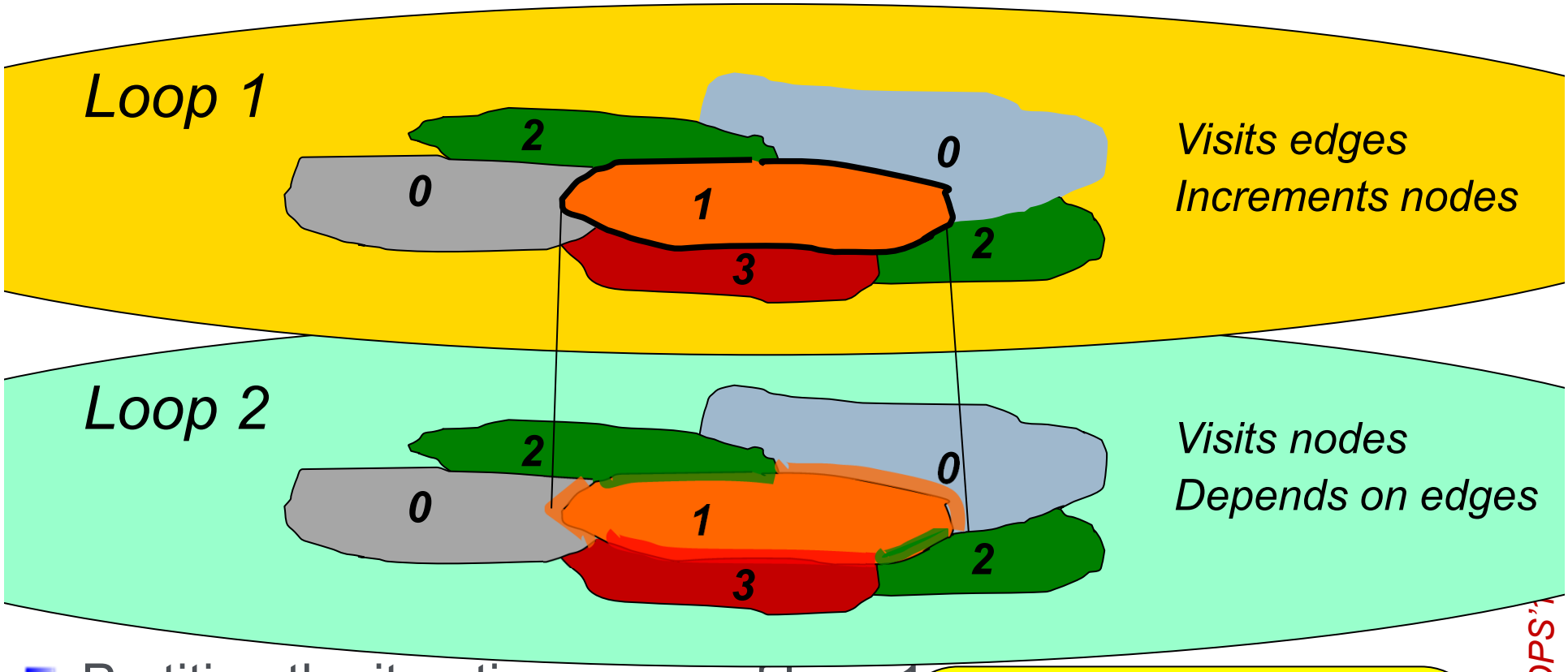
Sparse split tiling



- Partition the iteration space of loop 1
- Colour the partitions, execute the colours in order
- Project the tiles, using the knowledge that colour n can use data produced by colour n-1
- Thus, the tile coloured #1 ***grows*** where it meets colour #0
- And ***shrinks*** where it meets colours #2 and #3

Strout, Luporini et al, IPDPS'14

Sparse split tiling

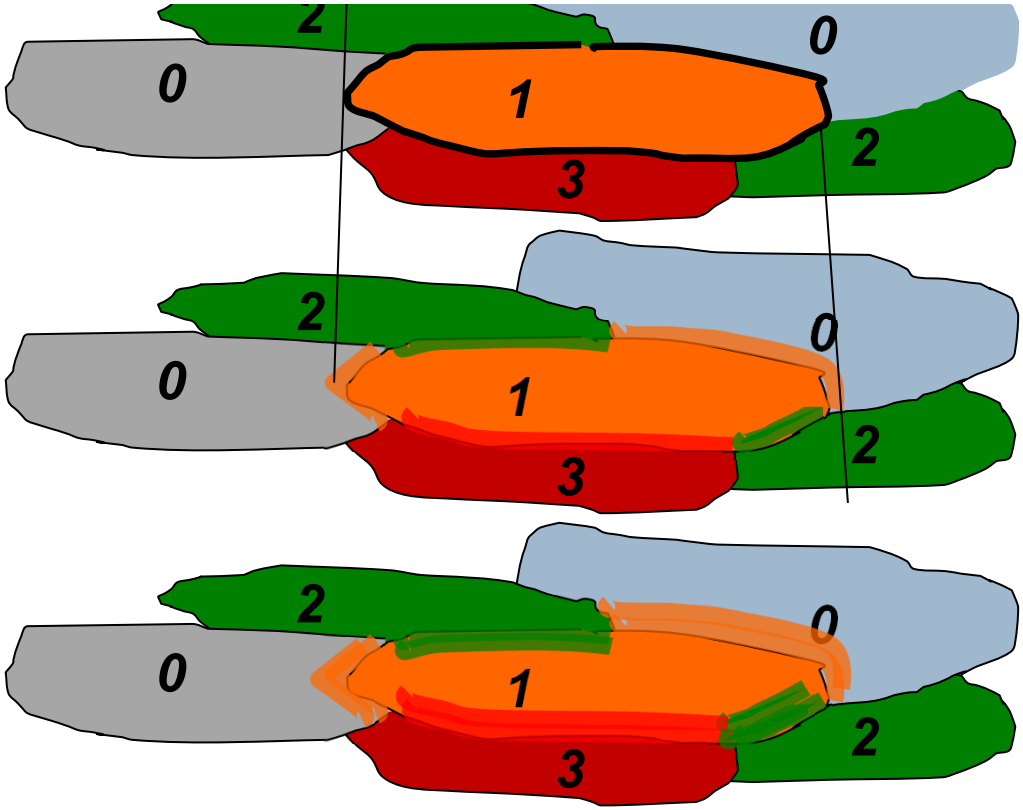


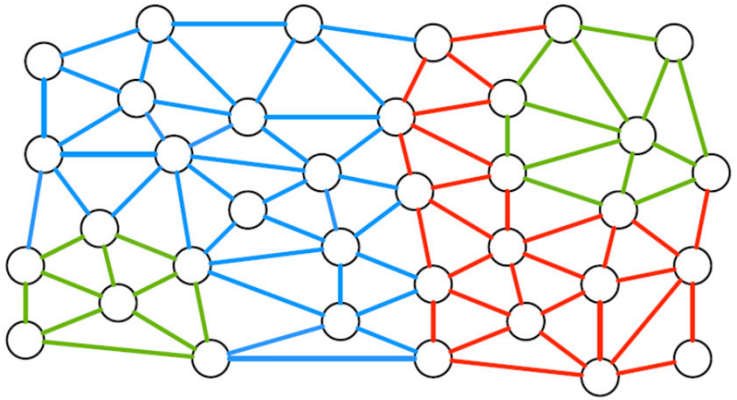
- Partition the iteration space of loop 1
- Colour the partitions
- Project the tiles, using the knowledg
  data produced by colour n-1
- Thus, the tile coloured #1 grows wh
- And shrinks where it meets colours #2 and #3

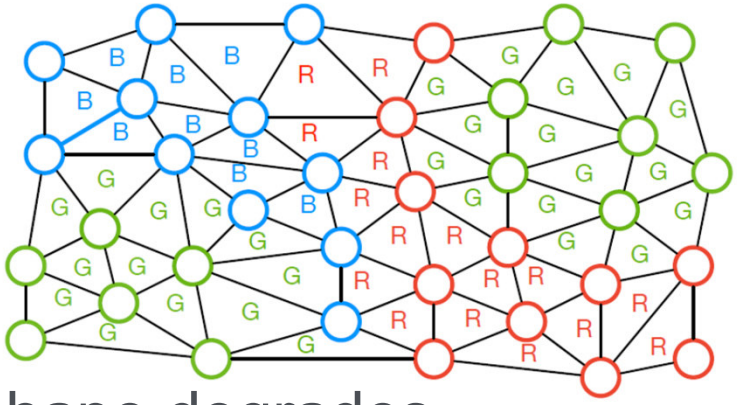
Inspector-executor: derive tasks and task graph from the mesh, ***at runtime***

Strout, Luporini et al, IPDPS'1

Tiles grow







- As we project the tiles forward, tile shape degrades
- Perimeter-volume ratio gets worse

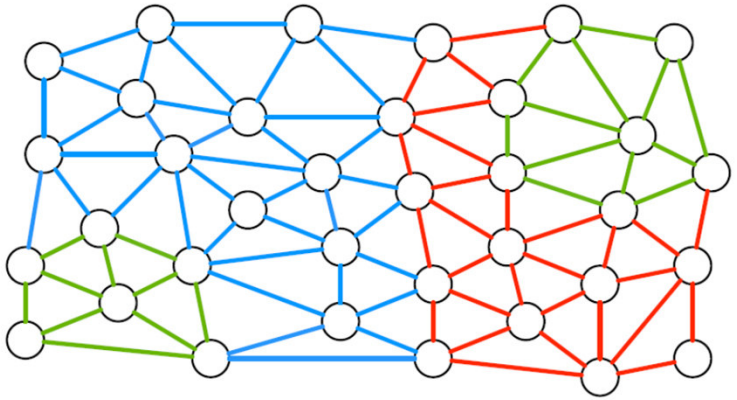
Strout, Luporini et al, IPDPS'14

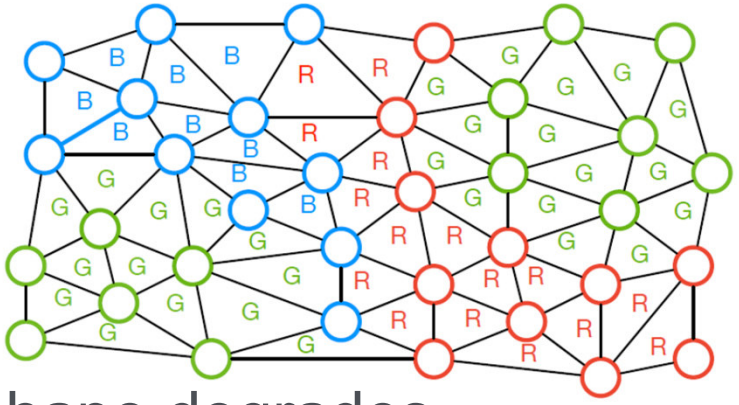
Tiles grow

Loop 1

Loop 2

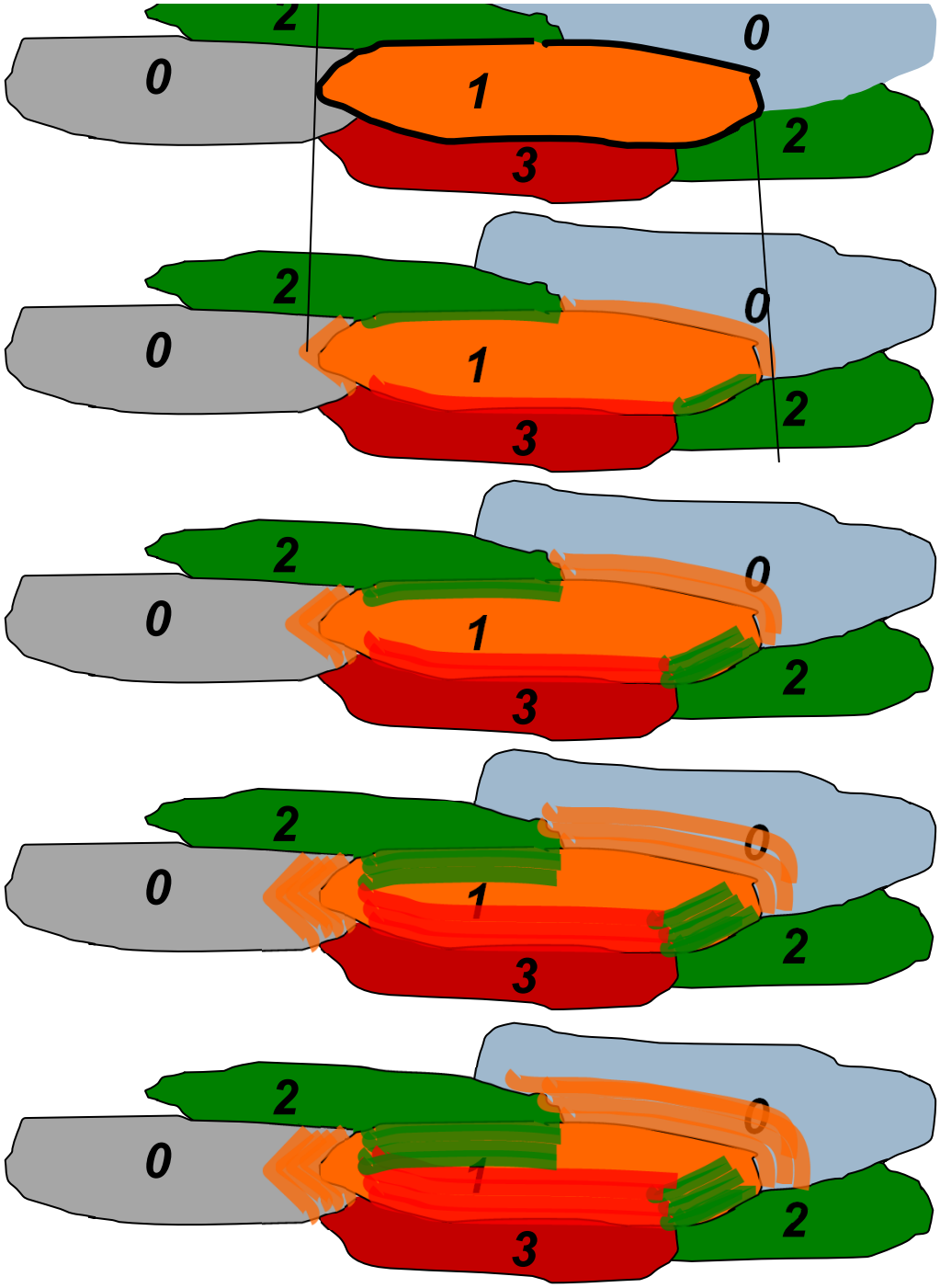
Loop 3



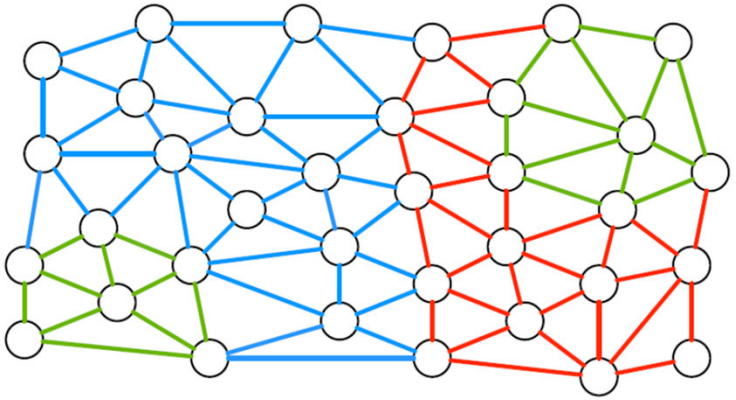


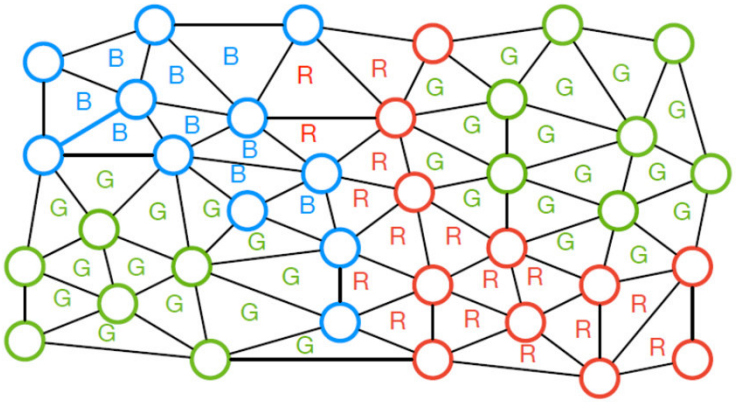
- As we project the tiles forward, tile shape degrades
- Perimeter-volume ratio gets worse
- We could partition Loop 1's data for the cache
- But Loop 2 and Loop 3 have different footprints
- So we rely on good (ideally space-filling-curve) numbering

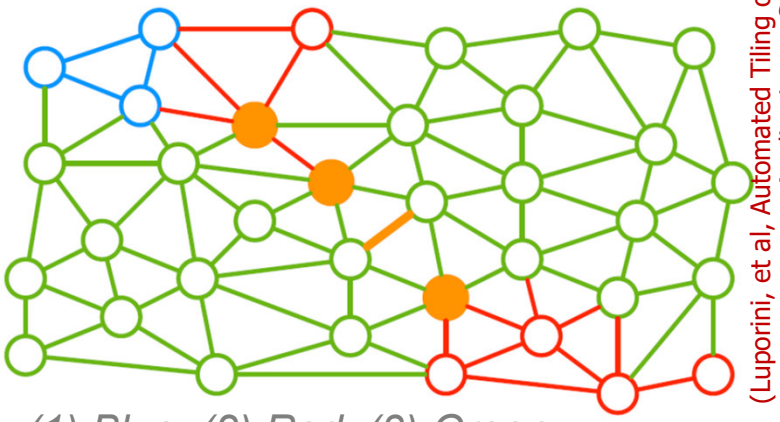
Strout, Luporini et al, IPDPS'14



Tiles can collide







(1) Blue, (2) Red, (3) Green

(Luporini, et al, Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling. ACM TOMS 2019)

```
while t <= T + 1e-12 and timestep < ntimesteps:
    if op2.MPI.COMM_WORLD.rank == 0 and timestep % self.output == 0:
        info("t = %f, (timestep = %d)" % (t, timestep))
    with loop_chain("main1",
                    tile_size=self.tiling_size,
                    num_unroll=self.tiling_uf,
                    mode=self.tiling_mode,
                    extra_halo=self.tiling_halo,
                    explicit=self.tiling_explicit,
                    use_glb_maps=self.tiling_glb_maps,
                    use_prefetch=self.tiling_prefetch,
                    coloring=self.tiling_coloring,
                    ignore_war=True,
                    log=self.tiling_log):
        # In case the source is time-dependent, update the time 't' here.
        if(self.source):
            with timed_region('source term update'):
                self.source_expression.t = t
                self.source = self.source_expression

        # Solve for the velocity vector field.
        self.solve(self.rhs_uh1, self.velocity_mass_asdat, self.uh1)
        self.solve(self.rhs_stemp, self.stress_mass_asdat, self.stemp)
        self.solve(self.rhs_uh2, self.velocity_mass_asdat, self.uh2)
        self.solve(self.rhs_u1, self.velocity_mass_asdat, self.u1)

        # Solve for the stress tensor field.
        self.solve(self.rhs_sh1, self.stress_mass_asdat, self.sh1)
        self.solve(self.rhs_utemp, self.velocity_mass_asdat, self.utemp)
        self.solve(self.rhs_sh2, self.stress_mass_asdat, self.sh2)
        self.solve(self.rhs_s1, self.stress_mass_asdat, self.s1)

    self.u0.assign(self.u1)
    self.s0.assign(self.s1)

    # Write out the new fields
    self.write(self.u1, self.s1, self.tofile and timestep % self.output == 0)

    # Move onto next timestep
    t += self.dt
    timestep += 1
```

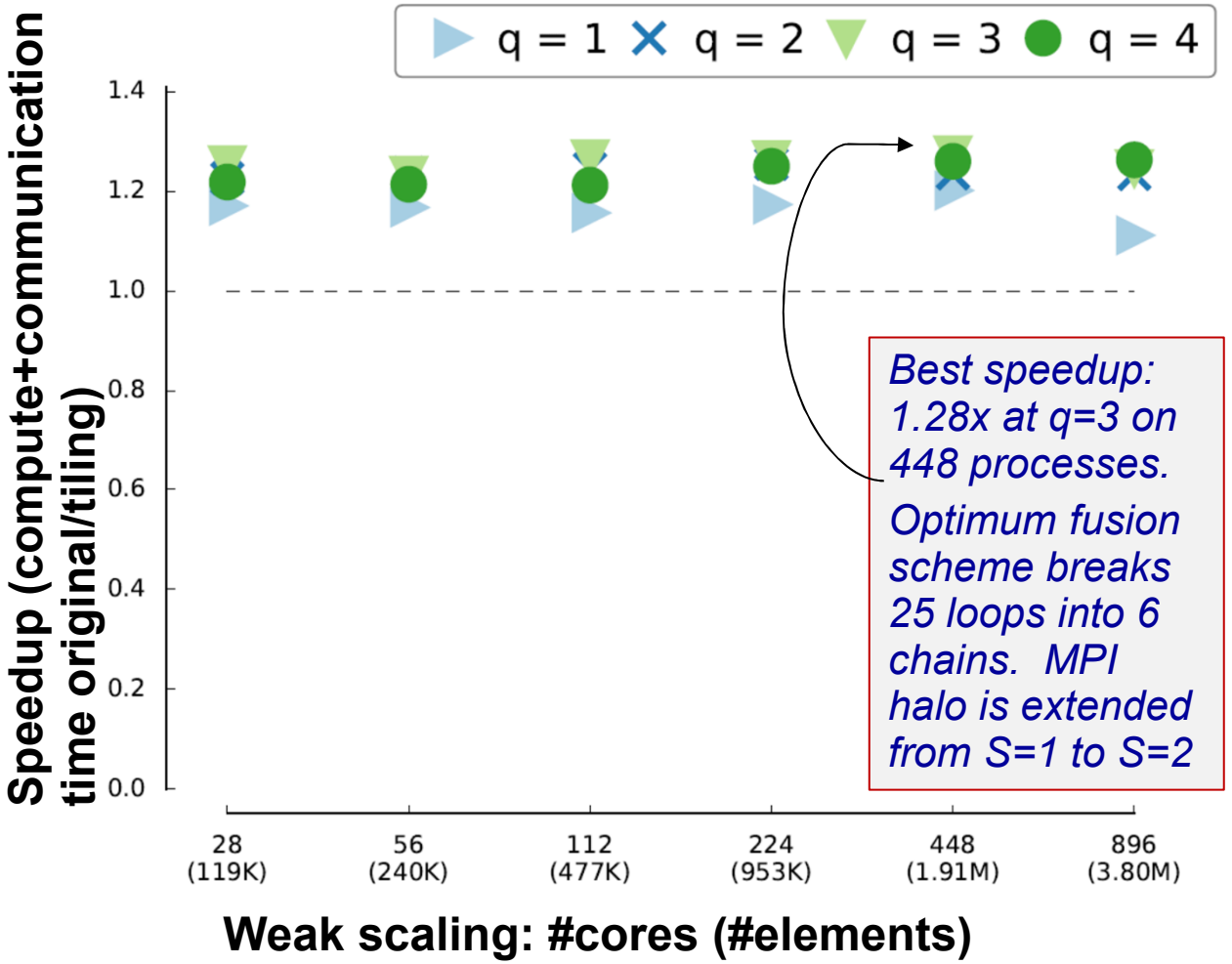
Loop chains

```
with loop_chain(tile_size=,….):
    # solve for velocity vector field
    self.solve(….);
    self.solve(….);
    self.solve(….);
    self.solve(….);
    # solve for stress tensor field
    self.solve(….);
    self.solve(….);
    self.solve(….);
    self.solve(….);
```

(25 op_par_loops per timestep, all tilable)

(Luporini, Lange, Jacobs, Gorman, Ramanujam, Kelly. Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling. ACM TOMS 2019 https://doi.org/10.1145/3302256)

- **Example: Seigen**
- Elastic wave solver
- 2d triangular mesh
- Velocity-stress formulation
- 4th-order explicit leapfrog timestepping scheme
- Discontinuous-Galerkin, order q=1-4
- 32 nodes, 2x14-core E5-2680v4, SGI MPT 2.14
- 1000 timesteps (ca.1.15s/timestep)



- Up to 1.28x speedup
- Inspection about as much time as 2 timesteps
- Using RCM numbering – space-filling curve should lead to better results

(ACM TOMS 2019)

- Can we automate interesting optimisations that would be hard to do by hand?

  - Second example:
    - Generalised loop-invariant code motion

    - (This optimisation has been implemented, automated, and re-implemented – and forms part of the standard distribution)

Recall:

```
t__ dat1, double const *__restrict__ dat0, int const *__restrict__ map0, int const *__restrict__ map1)
```

```
  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_t45 + form_t18[6 * form_ip + form_j0] * form_t38[form_k0_0] + form_t44;
          t2[1 + 24 * form_j0 + 2 * form_k0_0] = t2[1 + 24 * form_j0 + 2 * form_k0_0] + form_t18[6 * form_ip + form_k0_0] * 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);
  }
}
```

- 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

A simpler example:

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
          *det*W[i]);
}
```

- Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.
- The local assembly operation computes a small dense submatrix
- These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

Luporini, Varbenescu et al, ACM TACO/HiPEAC 2015

A simpler example:

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
          +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
          +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))
          *det*W[i]);
}
```

- Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.
- The local assembly operation computes a small dense submatrix
- These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

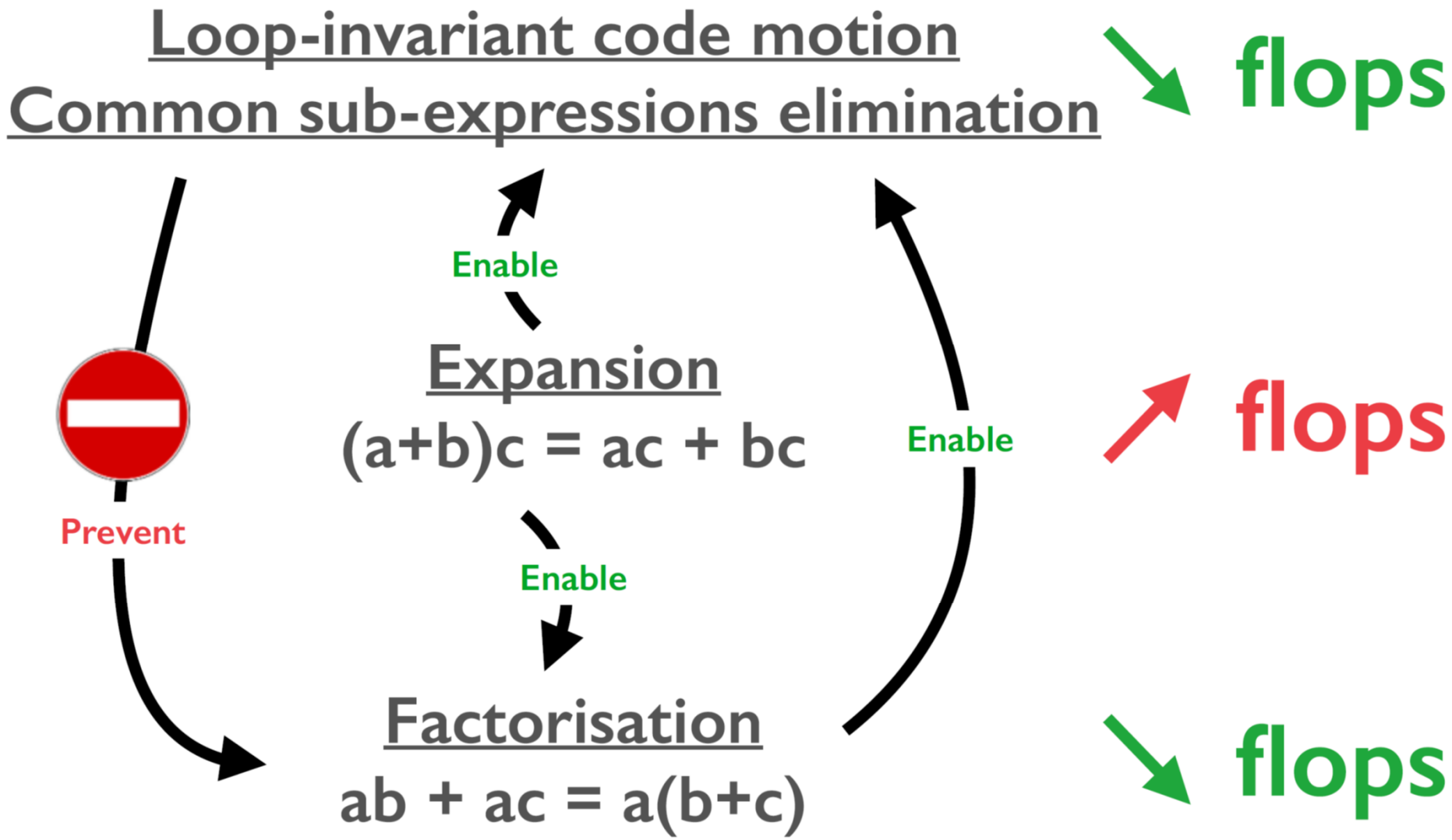
Generalised loop-invariant code motion:

```
void helmholtz(double A[3][4], double **coords) {
  #define ALIGN __attribute__((aligned(32)))
  // K, det = Compute Jacobian (coords)

  static const double W[3] ALIGN = {...}
  static const double X_D10[3][4] ALIGN = {{...}}
  static const double X_D01[3][4] ALIGN = {{...}}

  for (int i = 0; i<3; i++) {
    double LI_0[4] ALIGN;
    double LI_1[4] ALIGN;
    for (int r = 0; r<4; r++) {
      LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
      LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
    }
    for (int j = 0; j<3; j++)
      #pragma vector aligned
      for (int k = 0; k<4; k++)
        A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i];
  }
}
```

- Local assembly code for the Helmholtz problem after application of
  - padding,
  - data alignment,
  - Loop-invariant code motion
- In this example, sub-expressions invariant to j are identical to those invariant to k, so they can be precomputed once in the r loop

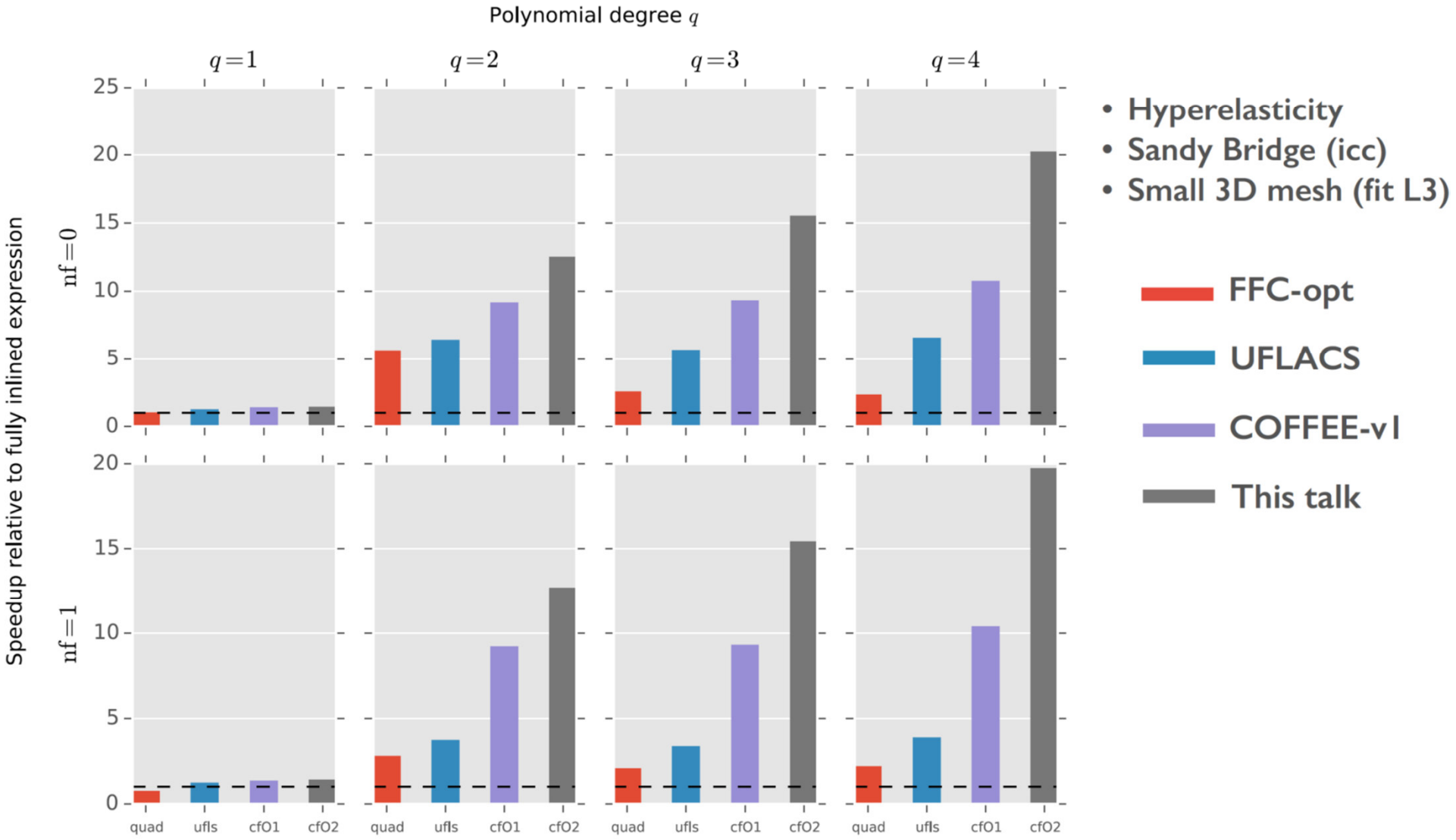
Luporini, Varbenescu et al, AC TACO/HiPEAC 2015

ARSENAL FOR REDUCING FLOPS



We formulate an ILP problem to find the best factorisation strategy

FOCUS ON HYPERELASTICITY



F. Luporini, D.A. Ham, P.H.J. Kelly. An algorithm for the optimization of finite element integration loops. ACM Transactions on Mathematical Software (TOMS), 2017).