Imperial College London

“Turing Tariff” Reduction: architectures, compilers and languages to break the universality barrier

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

This talk includes work done by or influenced by: David Ham (Imperial Maths), Andy Davison (Imperial), Lawrence Mitchell (Durham)
Gerard Gorman, Michael Lange (Imperial Earth Science Engineering – Applied Modelling and Computation Group)
Mike Giles, Gihan Mudalige, Istvan Reguly (Mathematical Inst, Oxford)
Fabio Luporini, Graham Markall, Florian Rathgeber, Francis Russell, George Rokos, Tianjiao Sun (Computing, Imperial)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)
Michelle Mills Strout, Chris Krieger, Cathie Olschanowsky (Colorado State University)
Carlo Bertolli, Doru Bercea (IBM Research), Richard Veras, Ram Ramanujam (Louisiana State University)
Doru Thom Popovici, Franz Franchetti (CMU), Karl Wilkinson (Capetown), Chris–Kriton Skylaris (Southampton)
Sajad Saeedi (Ryerson University), Luigi Nardi (Stanford/Lund University), Ridgway Scott (University of Chicago)

“Turing Tariff” Reduction: architectures, compilers and languages to break the universality barrier

- A little bit about my research
- A little bit of history
- A bit about how our **algorithms** textbooks are wrong/misguided
- A bit about how our **architecture** textbooks are wrong/misguided
- A bit about how our **compilers** textbooks are wrong/misguided
- The book I should be writing
- It’s all about skiing

- *This is not a research talk*
- *It’s a polemic*
- *Whose purpose is to provoke discussion*

Firedrake

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.

Latest commits to the Firedrake master branch on Github

Merge pull request #1520 from firedrakeproject/wence/feature/assemble-diagonal
Lawrence Mitchell authored at 22/10/2019, 09:14:34

tests: Check that getting diagonal of matrix works
Lawrence Mitchell authored at 21/10/2019, 13:04:04

matfree: Add getDiagonal method to implicit matrices
Lawrence Mitchell authored at 18/10/2019, 10:19:48

assemble: Add option to assemble diagonal of 2-form into Dat
Lawrence Mitchell authored at 18/10/2019, 10:08:37

Merge pull request #1509 from firedrakeproject/wence/patch-c-wrapper

firedrakeproject.org

Apps | Shareable Whitebo... | Startpage Search E... | Papers We Love | Tensor Decompositi... | The Conversation: I... | bookmarks

Fire

Documentation | Download | Team | Citir

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

Features:

- Expressive specification of any PDE using the Unified Form Languag
  Project.
- Sophisticated, programmable solvers through seamless coupling with
- 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 h
  elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.

Active team members


David Ham


Paul Kelly


Lawrence Mitchell


Thomas Gibson


Tianjiao (TJ) Sun


Miklós Homolya


Andrew McRae


Colin Cotter


Rob Kirby


Koki Sagiyama

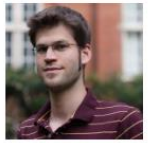
Former team members


Fabio Luporini


Alastair Gregory


Michael Lange

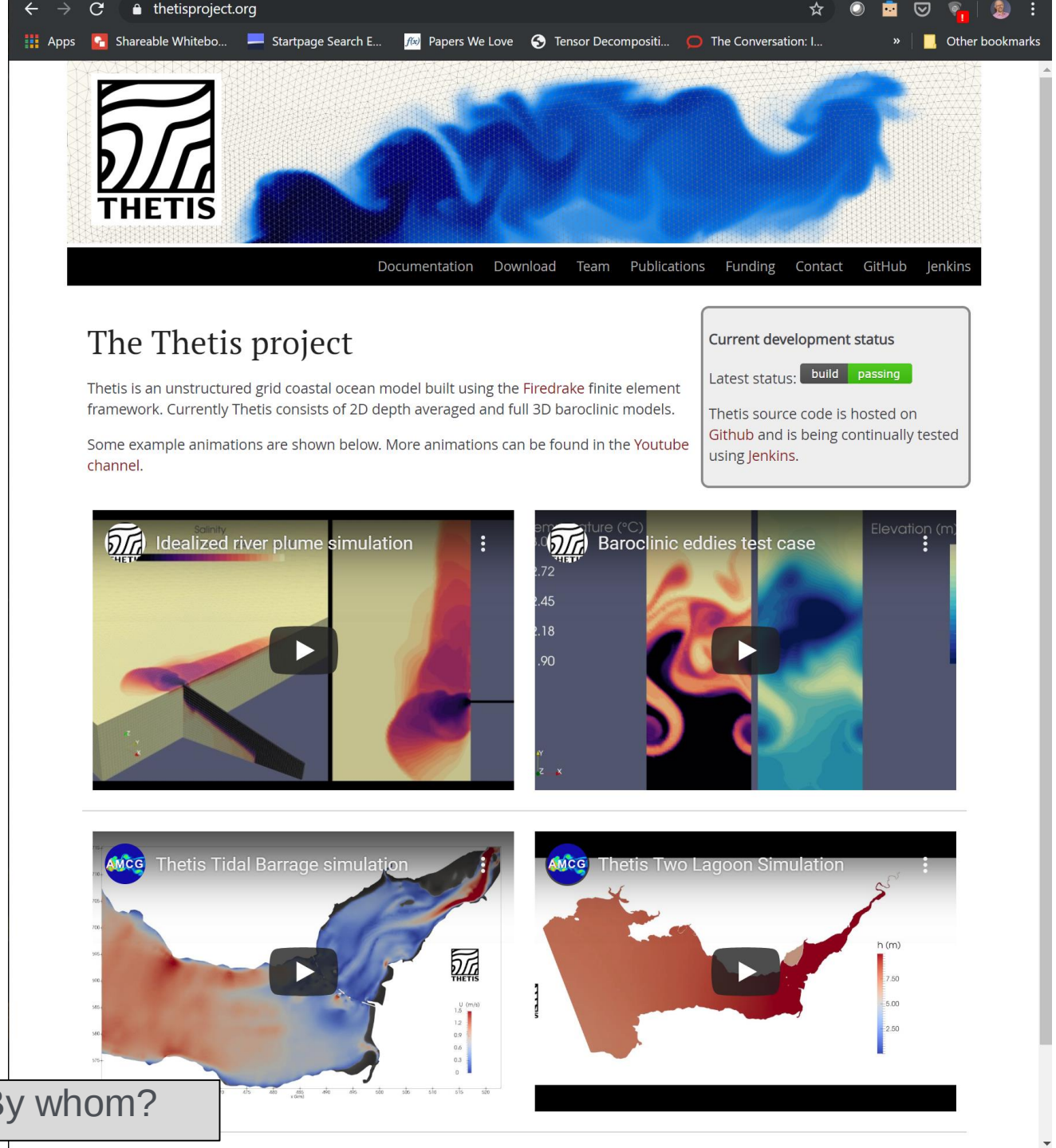

Simon Funke


Florian Rathgeber

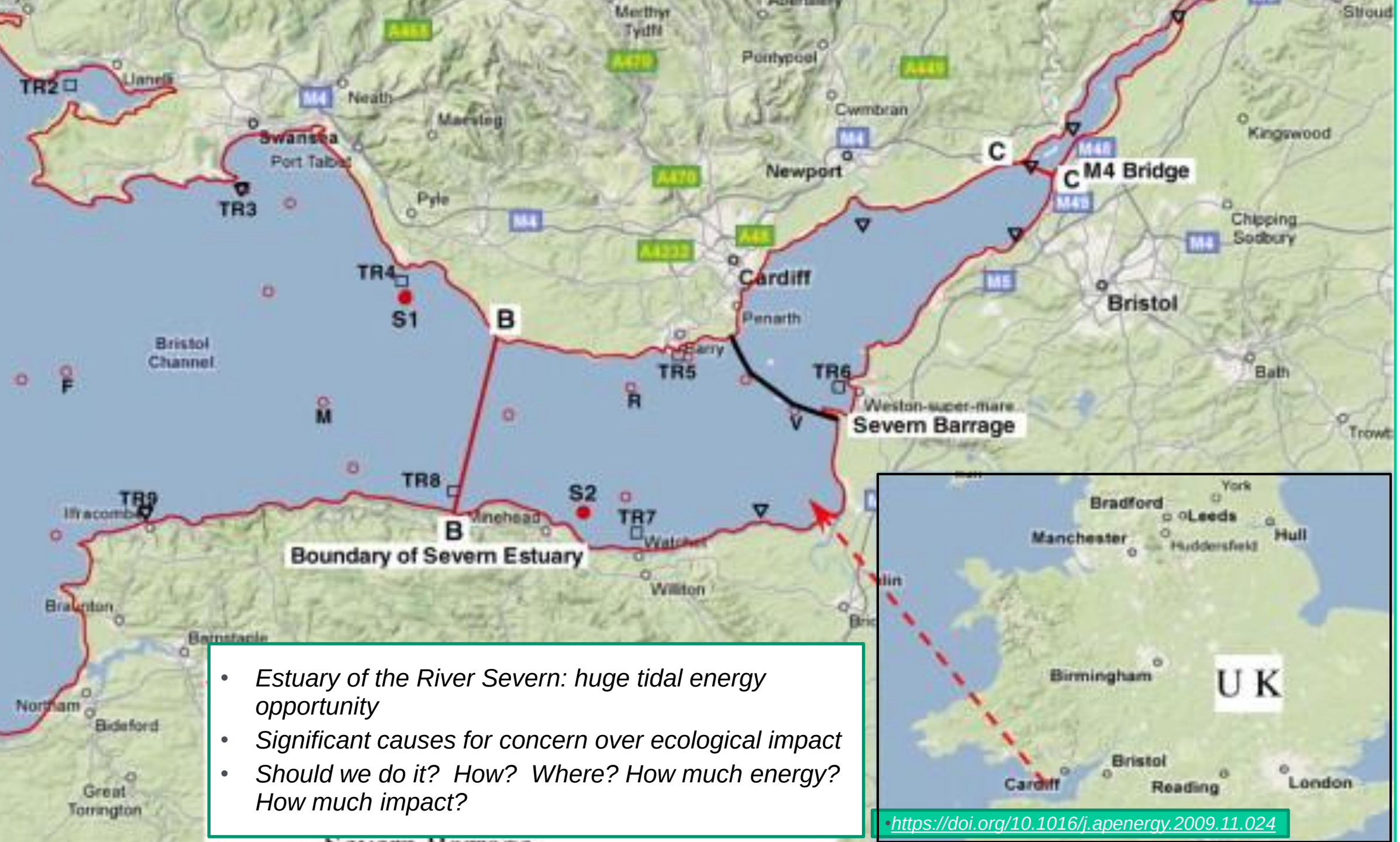

Doru Bercea


Graham Markall

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

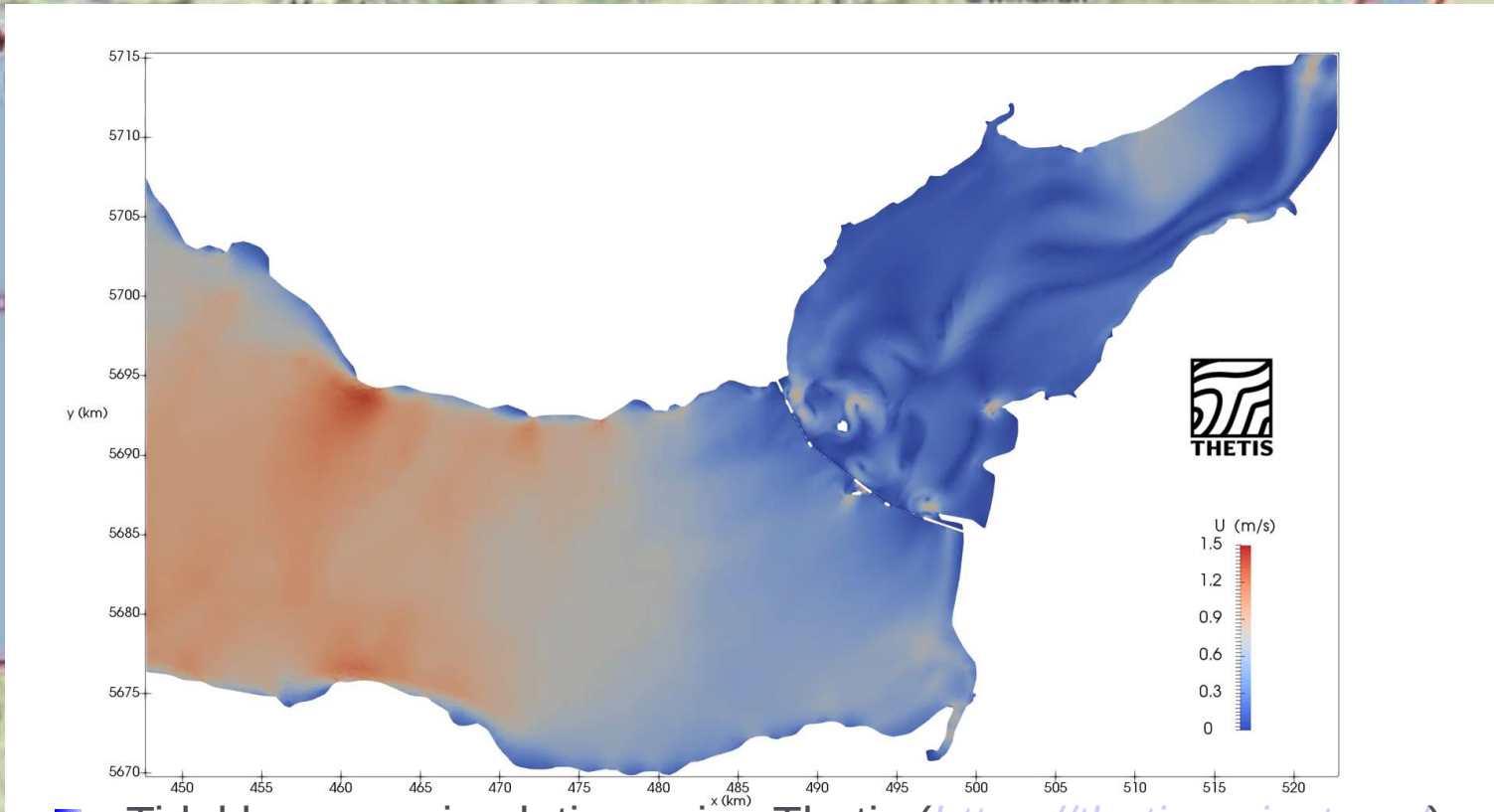


- What is it used for? By whom?



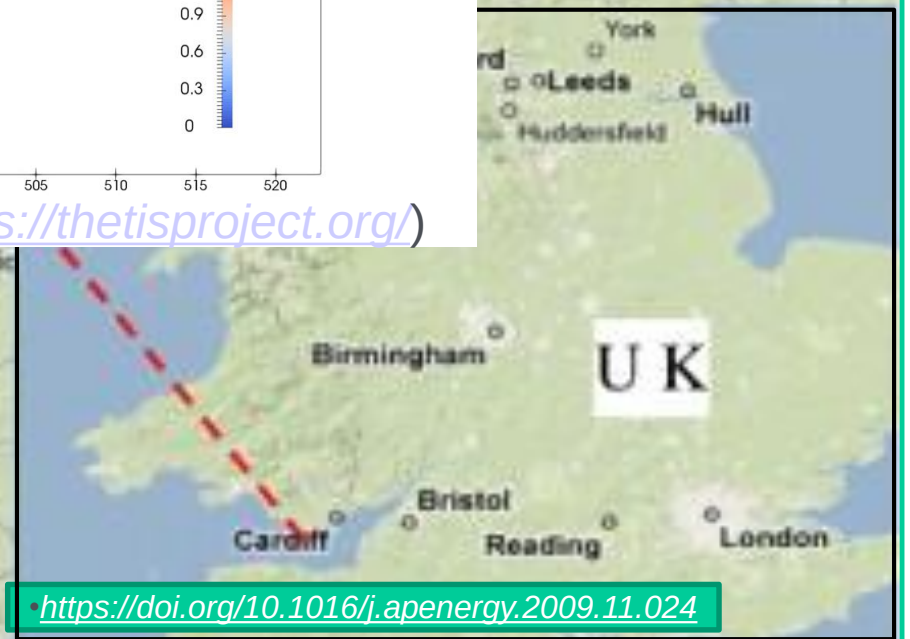
- *Estuary of the River Severn: huge tidal energy opportunity*
- *Significant causes for concern over ecological impact*
- *Should we do it? How? Where? How much energy? How much impact?*

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



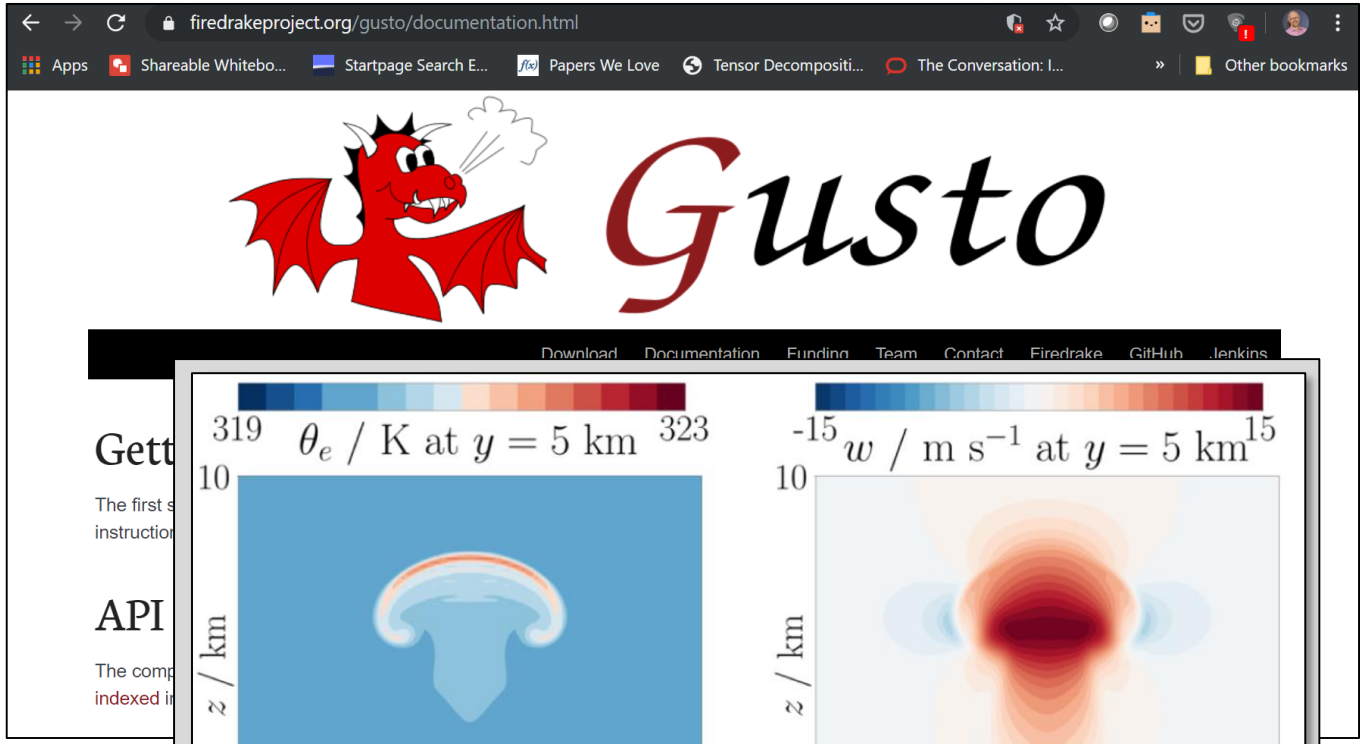
Tidal barrage simulation using Thetis (https://thetisproject.org/)

- *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

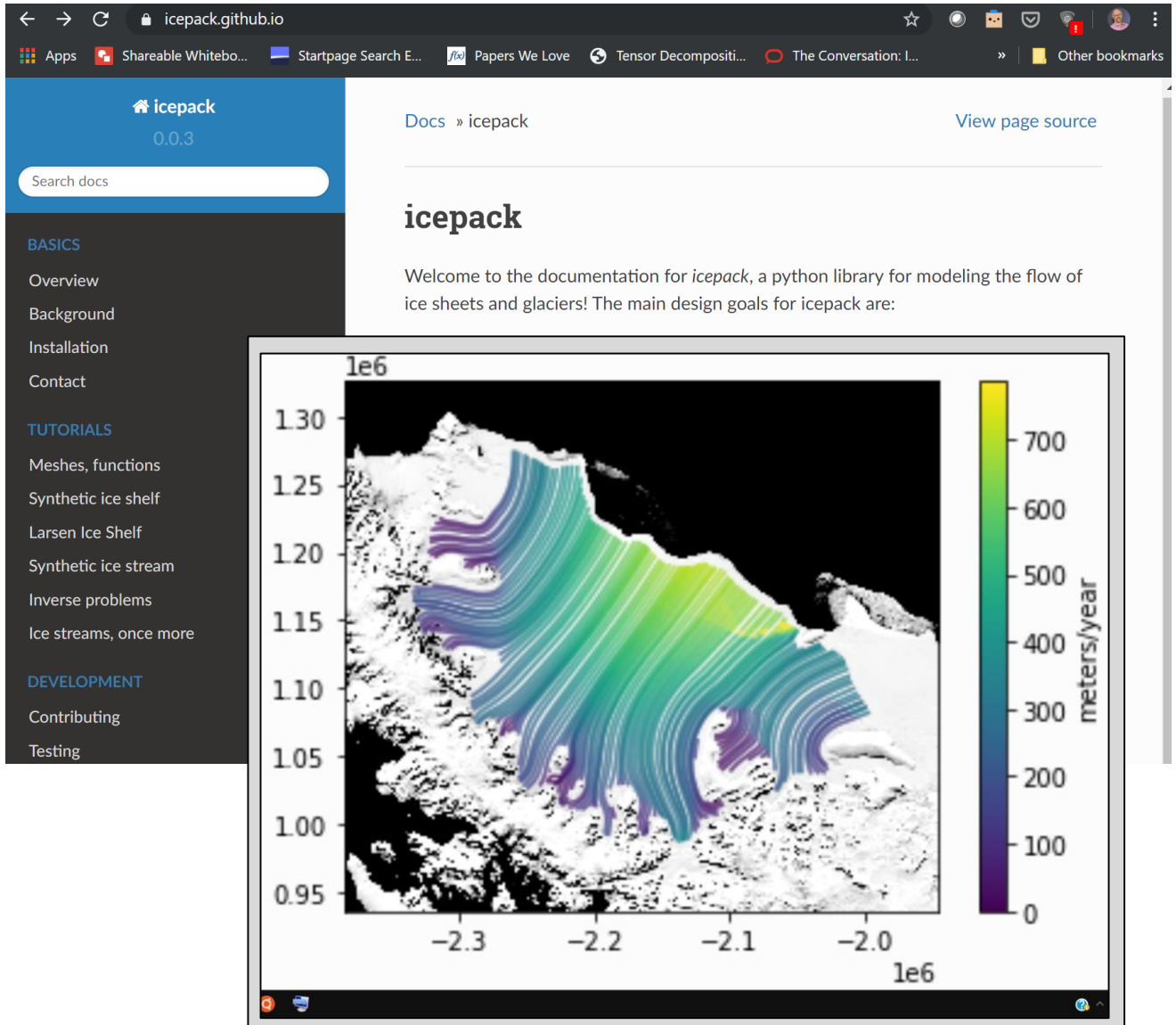
- Firedrake is used in:
  - **Gusto**: atmospheric modelling framework being used to prototype the next generation of weather and climate simulations for the UK Met Office



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?

- 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

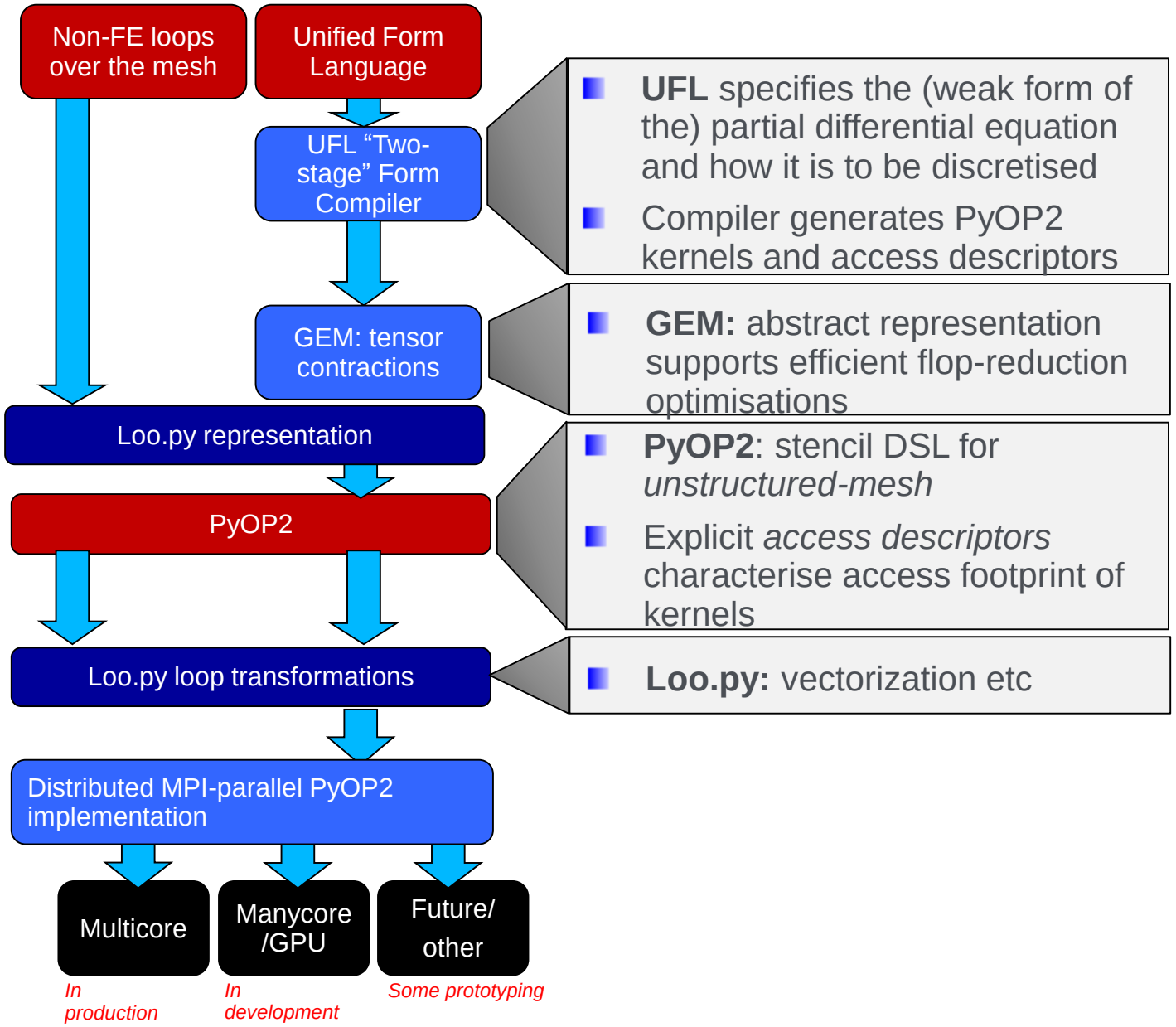


Larsen ice shelf model, from the Icepack tutorial by Daniel Shapero (https://icepack.github.io/icepack.demo.02-larsen-ice-shelf.html)

- What is it used for? By whom?

Firedrake: a finite-element framework

- Automates the finite element method for solving PDEs
- Alternative implementation of FEniCS language, 100% Python using runtime code generation



Rathgeber, Ham, Mitchell et al, ACM TOMS 2016

https://www.firedrakeproject.org/

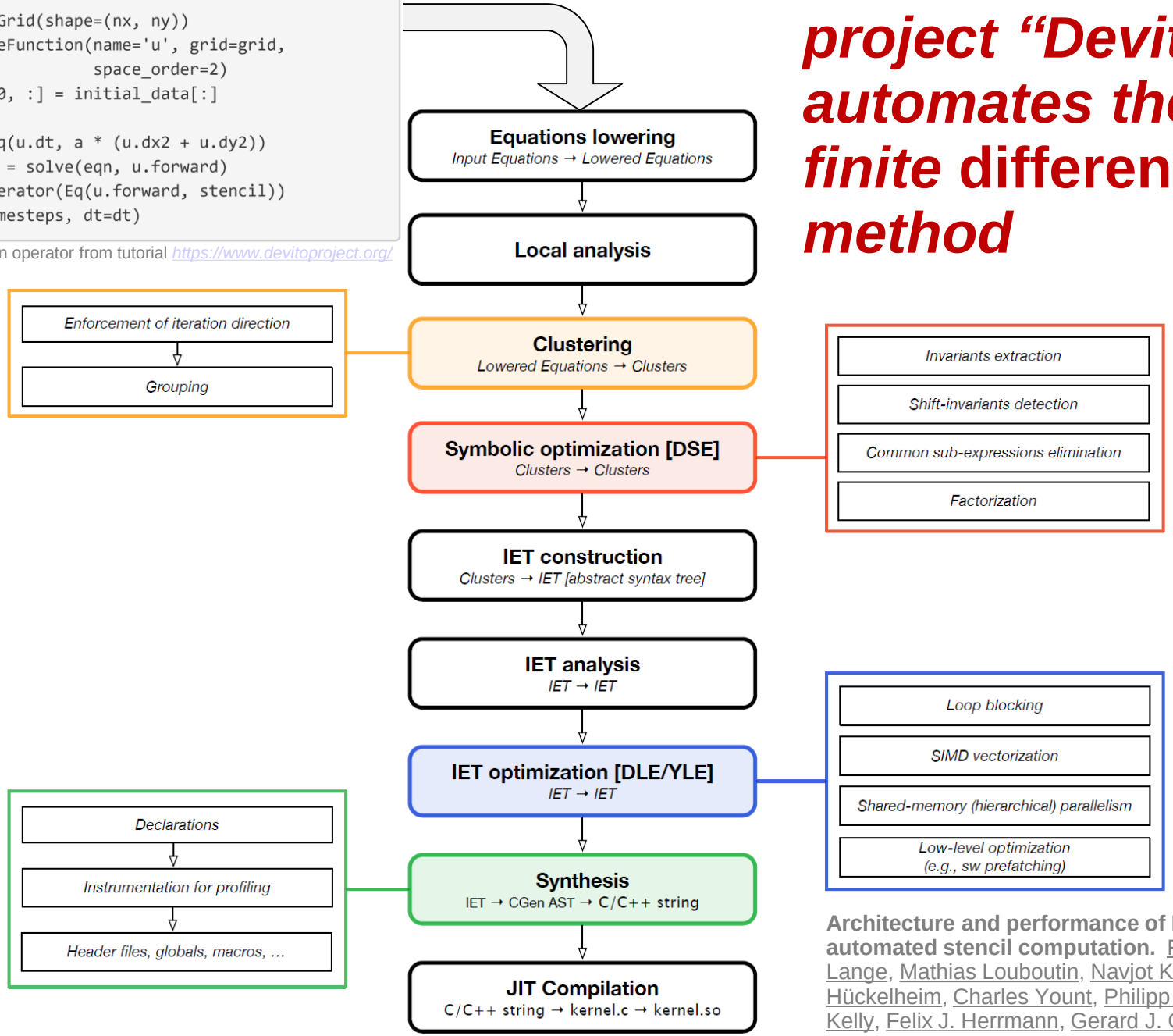
Firedrake's sibling project "Devito" automates the finite difference method

```
from devito import *

grid = Grid(shape=(nx, ny))
u = TimeFunction(name='u', grid=grid,
                 space_order=2)
u.data[0, :] = initial_data[:]

eqn = Eq(u.dt, a * (u.dx2 + u.dy2))
stencil = solve(eqn, u.forward)
op = Operator(Eq(u.forward, stencil))
op(t=timesteps, dt=dt)
```

2D diffusion operator from tutorial https://www.devitoproject.org/

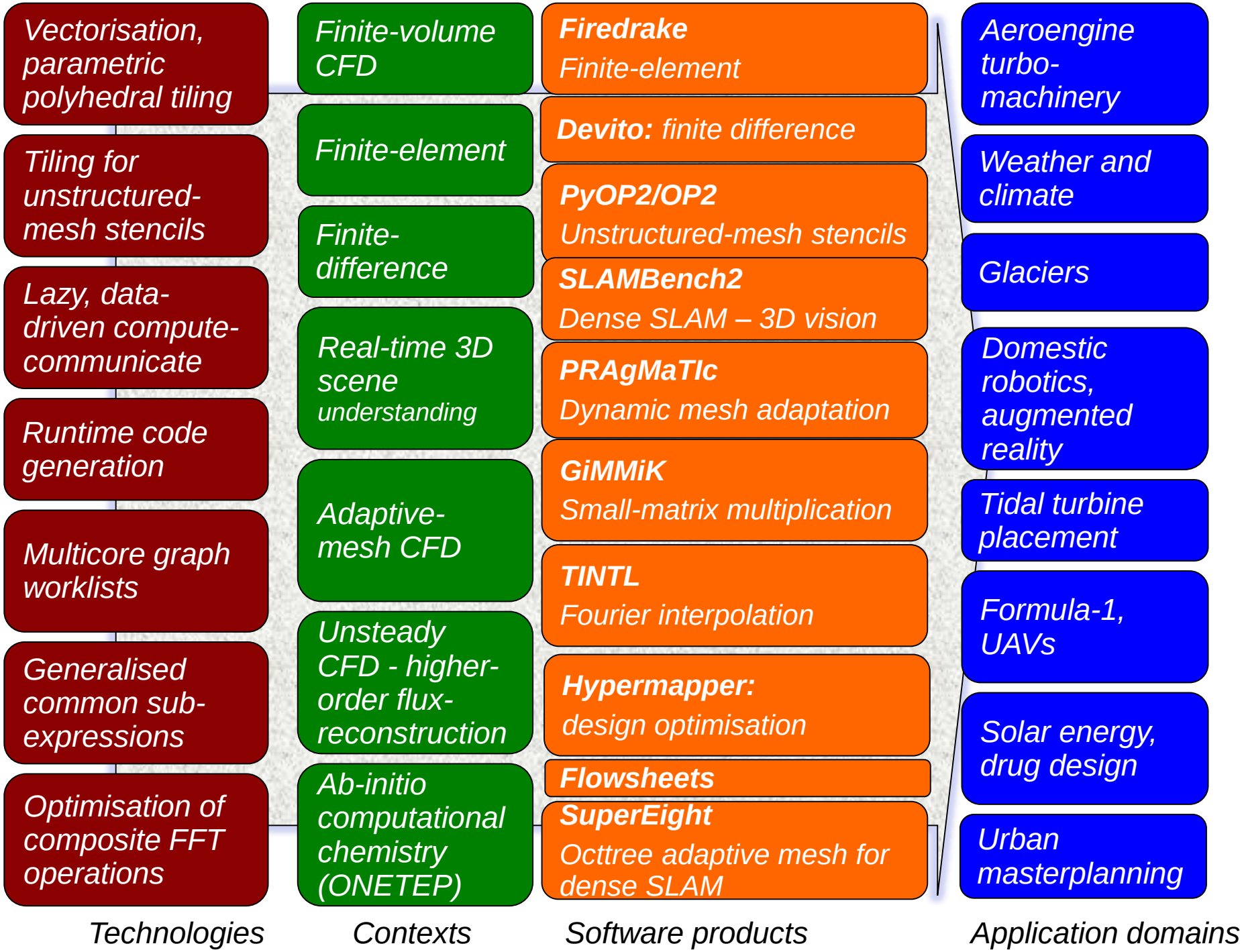


Architecture and performance of Devito, a system for automated stencil computation. Fabio Luporini, Michael Lange, Mathias Louboutin, Navjot Kukreja, Jan Hückelheim, Charles Yount, Philipp Witte, Paul H. J. Kelly, Felix J. Herrmann, Gerard J. Gorman. ACM TOMS (accepted). https://arxiv.org/abs/1807.03032

https://www.devitoproject.org/

Domain-specific optimisation

Targetting MPI, OpenMP, OpenCL, Dataflow/ FPGA, from HPC to mobile, embedded and wearable



Feynmann: plenty of room at the bottom

Miniaturizing the computer

I don't know how to do this on a small scale in a practical way, but I do know that computing machines are very large; they fill rooms. Why can't we make them very small, make them of little wires, little elements—and by little, I mean little. For instance, the wires should be 10 or 100 atoms in diameter, and the circuits should be a few thousand angstroms across.

https://en.wikipedia.org/wiki/There's_Plenty_of_Room_at_the_Bottom

(1959, talk at the American Physical Society)

Feynmann: plenty of room at the bottom

Miniaturizing the computer

I don't know how to do this on a small scale in a practical way, but I do know that computing machines are very large; they fill rooms. Why can't we make them very small, make them of little wires, little elements—and by little, I mean little. For instance, the wires should be 10 or 100 atoms in diameter, and the circuits should be a few thousand angstroms across.

- >60 years of exponential progress since then
- We're much closer to such limits
- Much debate about where they really lie
- What is clear is that we're a lot closer
- We are confronted more and more with fundamental physical concerns
- Particularly wrt communication latency, bandwidth and energy.

(1959, talk at the American Physical Society)

https://en.wikipedia.org/wiki/There's_Plenty_of_Room_at_the_Bottom

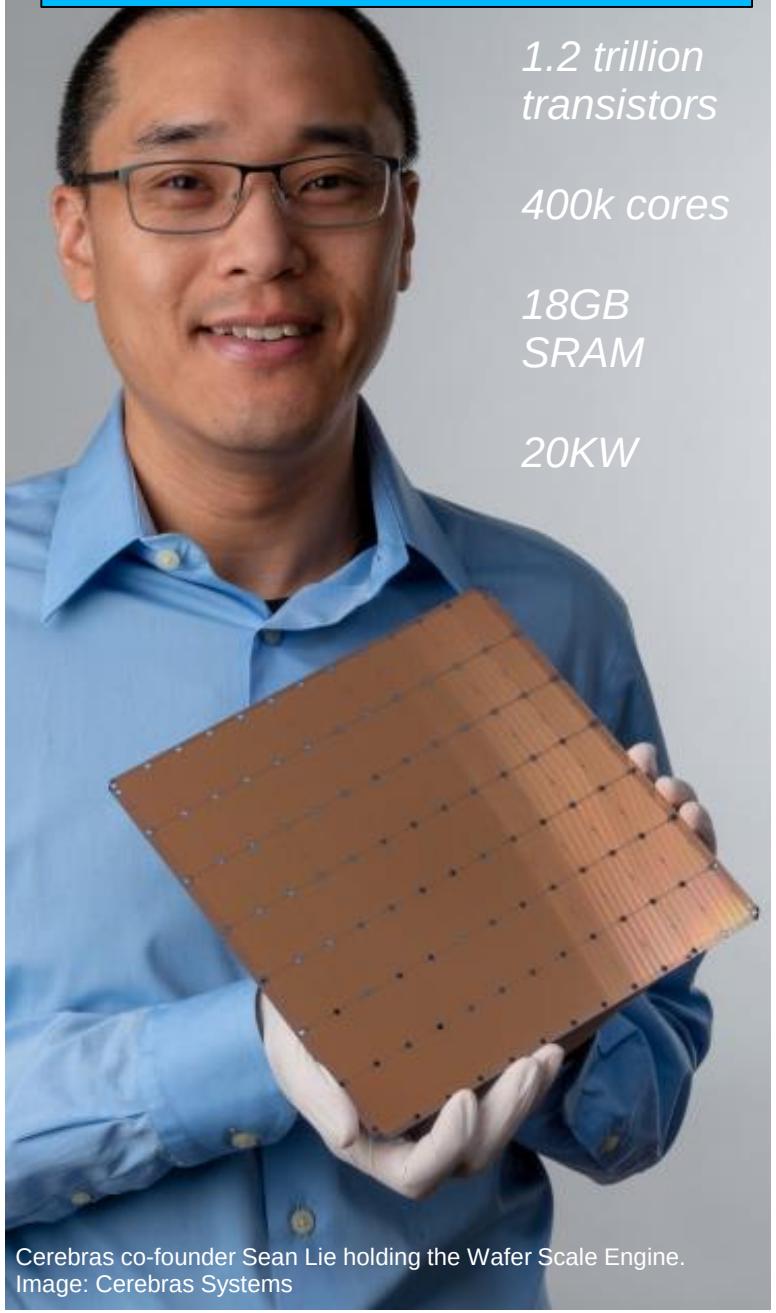
Ferranti Pegasus (1956-59)

Ferranti Computer Systems Ltd Pegasus valve computer circuit board, c. 1964

https://blog.sciencemuseum.org.uk/the-pegasus-computer/

- Cf Moore's Law: "circuit density doubles every 18 months"
- 60 years =40x18months
- So Moore's Law would predict 2^{40}= 10^{12} increase

Cerebras CS-1 (2020)



1.2 trillion transistors

400k cores

18GB SRAM

20KW

Cerebras co-founder Sean Lie holding the Wafer Scale Engine.
Image: Cerebras Systems

Algorithmic complexity and scheduling

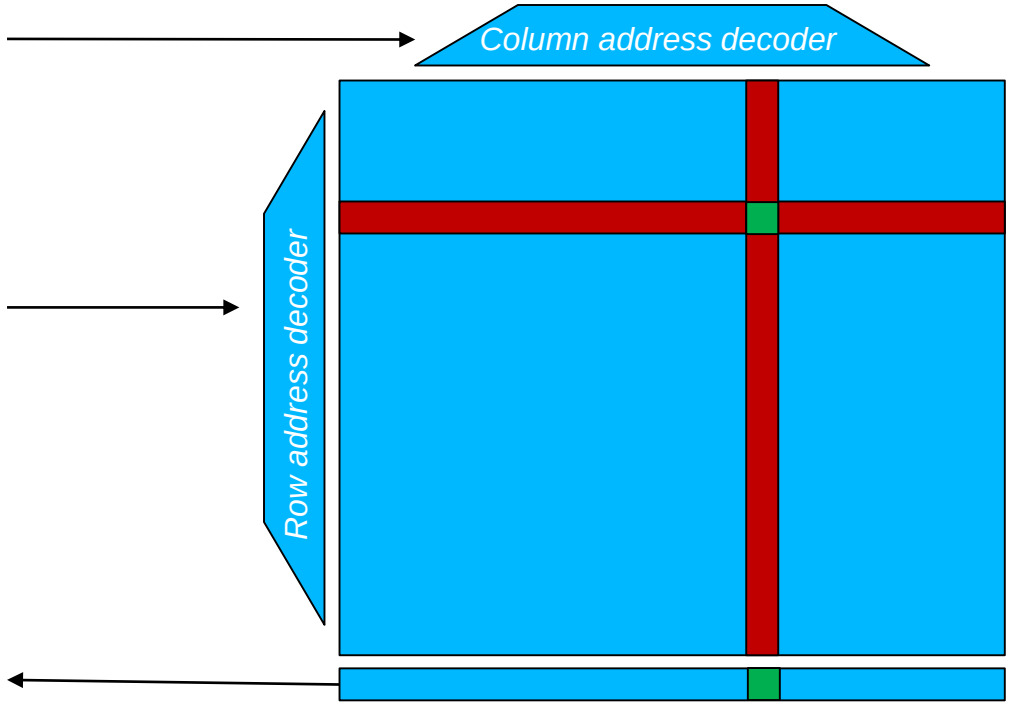
- Suppose there were no more room at the bottom
- How should that change how we think?
- About **algorithms?**

- We teach that access to a hash table is O(1), ie independent of the size of the hash table
  - And that it doesn't matter how you want to access your hash table, it's *still* O(1)

Algorithmic complexity and scheduling

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **algorithms?**

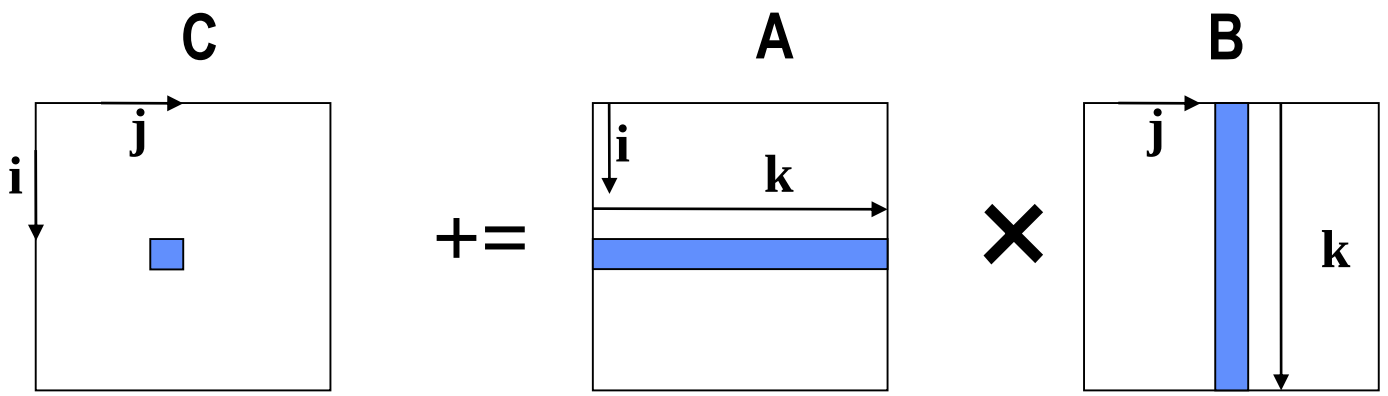
- We teach that access to a hash table is O(1), ie independent of the size of the hash table
  - But the hash table is implemented using a RAM distributed 3D space
  - So wire length increases with RAM size
  - And caching doesn't help since access is randomised



Algorithmic complexity and scheduling

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **algorithms?**

- We know that matrix-matrix multiply is $O(n^3)$
  - But in a deep memory hierarchy, access time depends on reuse distance
  - So naïve "for i for j for k" loop nest suffers reuse access latency that grows with N
  - Anecdotally, execution time $\sim O(n^5)$




```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  for (k=0; k<N; k++)
   C[i][j]+=A[i][k]*B[k][j]
```

- Each row of A is reused for a series of dot-products
- But if the cache is too small, it doesn't fit

Algorithmic complexity and scheduling

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **algorithms?**

```
for (kk = 0; kk < N; kk += S)
  for (jj = 0; jj < N; jj += S)
    for (i = 0; i < N; i++)
      for (k = kk; k < min(kk+S,N); k++)
        for (j = jj; j < min(jj+S, N); j++)
          C[i][j] += A[i][k] * B[k][j];
```

- Tiling for cache bounds the reuse distance so that reused submatrix fits in cache
- With a deep hierarchy we have to do this at every level of the cache, *recursively*
- Doing this leads to a big-O performance improvement
- Finding schedules with good locality is really an *algorithmic* challenge

Alpern, B., Carter, L., Feig, E. et al. The uniform memory hierarchy model of computationAlgorithmica (1994) 12: 72.

Turing ~~tax~~ tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture ?**

- 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**

Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute is the original Turing tariff

Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute is the original Turing tariff
- FPGAs pay Turing tariffs in the reconfigurable fabric

Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute is the original Turing tariff
- FPGAs pay Turing tariffs in the reconfigurable fabric
- Registers are a Turing Tariff
  - Because if we know the program's dataflow, we can use wires and latches to pass data from functional unit to functional unit
- Memory
  - But if we can stream data from where it's produced to where it's used, maybe we don't need so much RAM?

Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute is the original Turing tariff
- FPGAs pay Turing tariffs in the reconfigurable fabric
- Registers are a Turing Tariff
  - Because if we know the program's dataflow, we can use wires and latches to pass data from functional unit to functional unit
- Memory
  - But if we can stream data from where it's produced to where it's used, maybe we don't need so much RAM?
- Cache
  - If we know exactly when the reuse will occur, we can program movement to and from local fast memory explicitly

Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute is the original Turing tariff
- FPGAs pay Turing tariffs in the reconfigurable fabric
- Registers are a Turing Tariff
  - Because if we know the program's dataflow, we can use wires and latches to pass data from functional unit to functional unit
- Memory
  - But if we can stream data from where it's produced to where it's used, maybe we don't need so much RAM?
- Cache
  - If we know exactly when the reuse will occur, we can program movement to and from local fast memory explicitly
- Floating-point arithmetic:
  - If we know the dynamic range of expected values…

Turing tariffs – how architects pay

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- Fetch-execute, decode
- Registers, forwarding
- Dynamic instruction scheduling, cracking, packing, renaming
- Cache tags
- Cache blocks
- Cache coherency
- Prefetching
- Branch prediction
- Speculative execution
- Address translation
- Store-to-load forwarding, write combining, address decoding, ECC, DRAM refresh
- Mis-provisioning: unused bandwidth, unusable FLOPs, under-used accelerators

Basically the whole computer architecture textbook

How architects avoid Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **architecture?**

- SIMD: amortise fetch-execute over a vector or matrix of operands
- VLIW, EPIC, register rotation
- Macro-instructions: FMA, crypto, conflict-detect, custom ISAs
- Streaming dataflow: FPGAs, CGRAs
- Systolic arrays
- Circuit switching instead of packet switching
- DMA
- Predication
- Long cache lines
- Non-temporal loads/stores, explicit prefetch instructions
- Scratchpads
- Multi-threading
- Message passing

How *compilers* avoid Turing tariffs

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **compilers?**

Generating code to avoid the need for interpretive mechanisms in hardware:

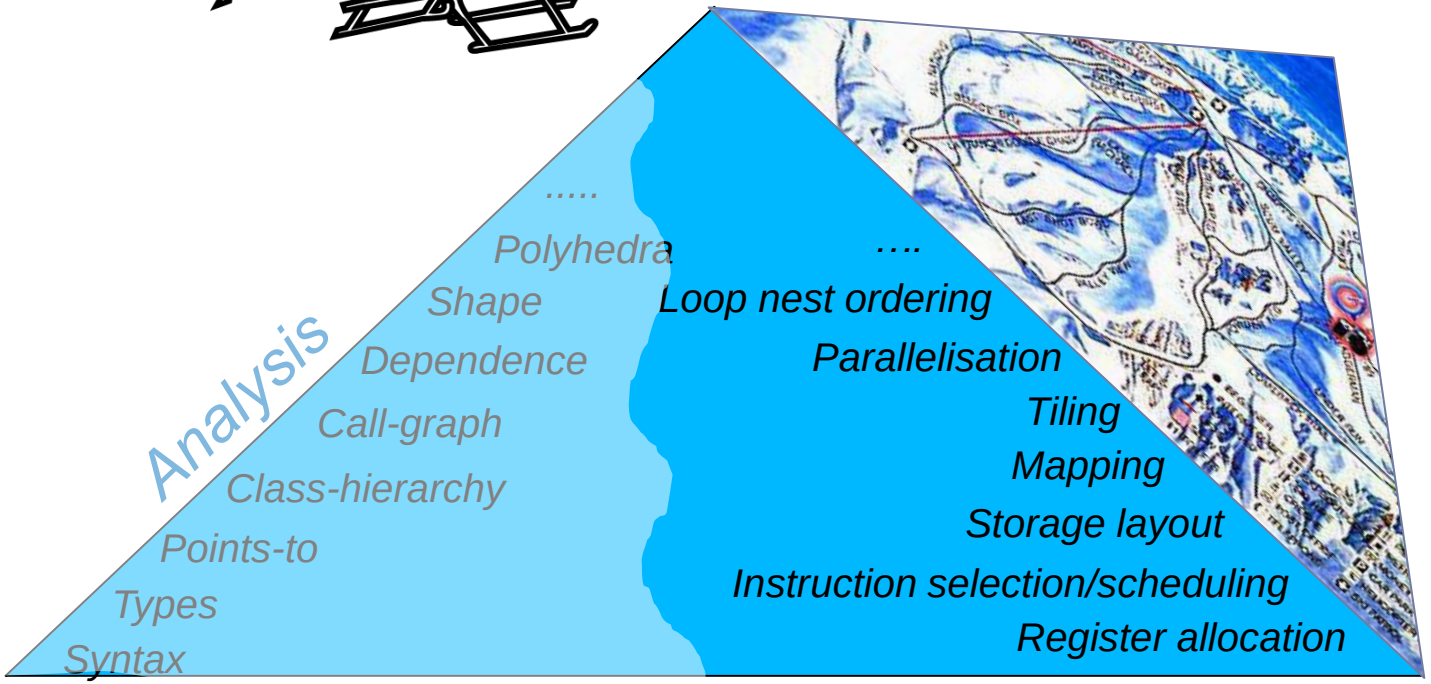
- Vectorisation
- Static instruction scheduling
- Offloading
- Predication
- Message aggregation
- Synchronisation minimization

Generating code that is specialized for a specific purpose:

- Function inlining, type disambiguation, object inlining
- Specialisation: metaprogramming, JIT, metatracing

- Suppose there were no more room at the bottom
- How should that change how we think?
- About **compilers?**

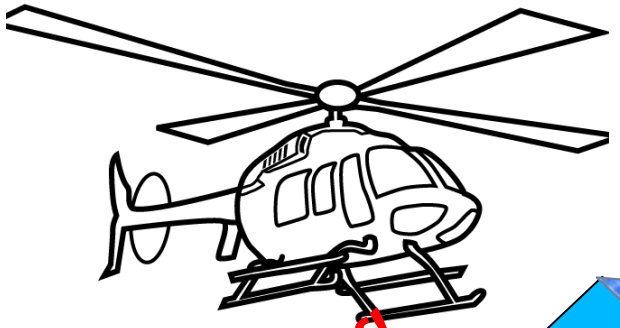
- Compilation is like skiing



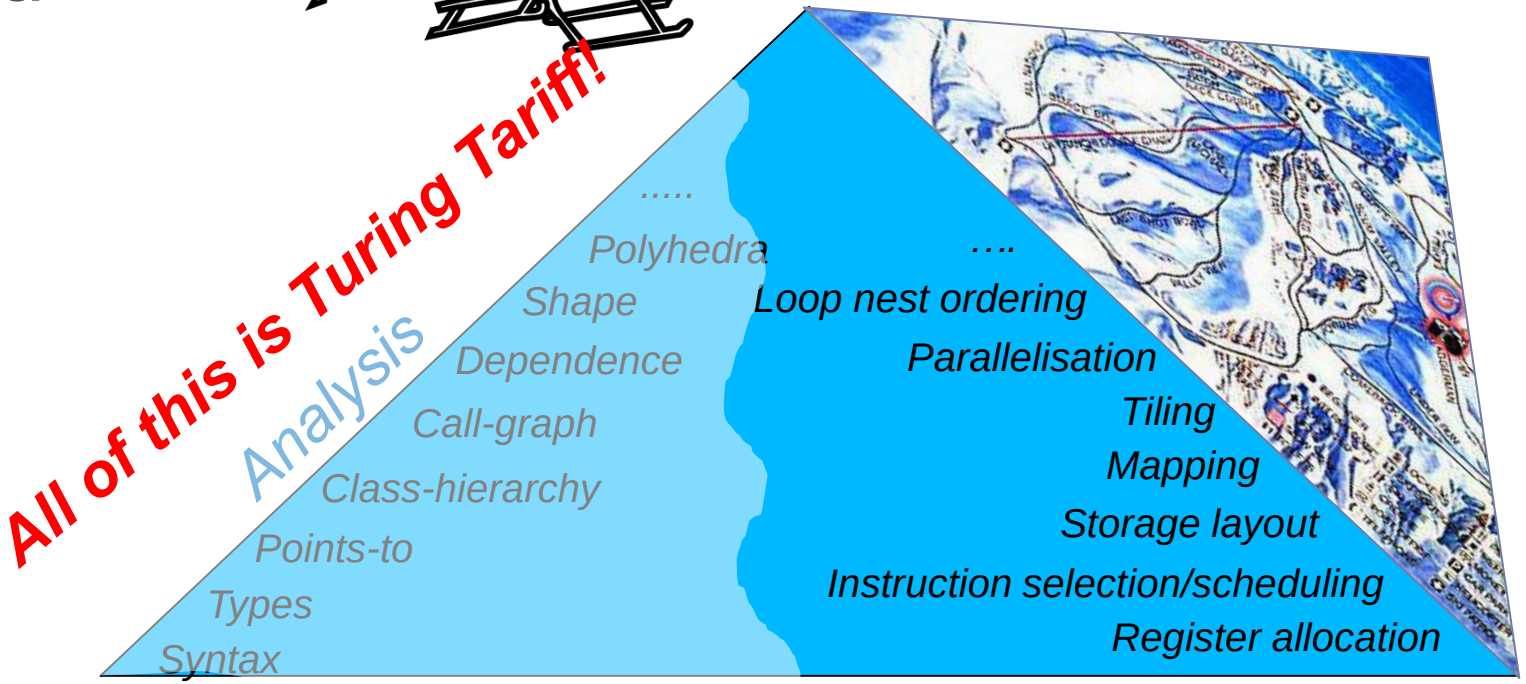
- Analysis is not always the interesting part....
- It's more fun the higher you start!

http://www.nikkiemcdade.com/subFiles/2DExamples.html

- General-purpose programming languages make you pay Turing tariffs!



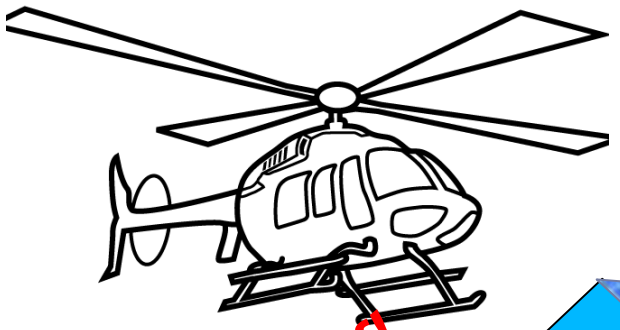
- Compilation is like skiing



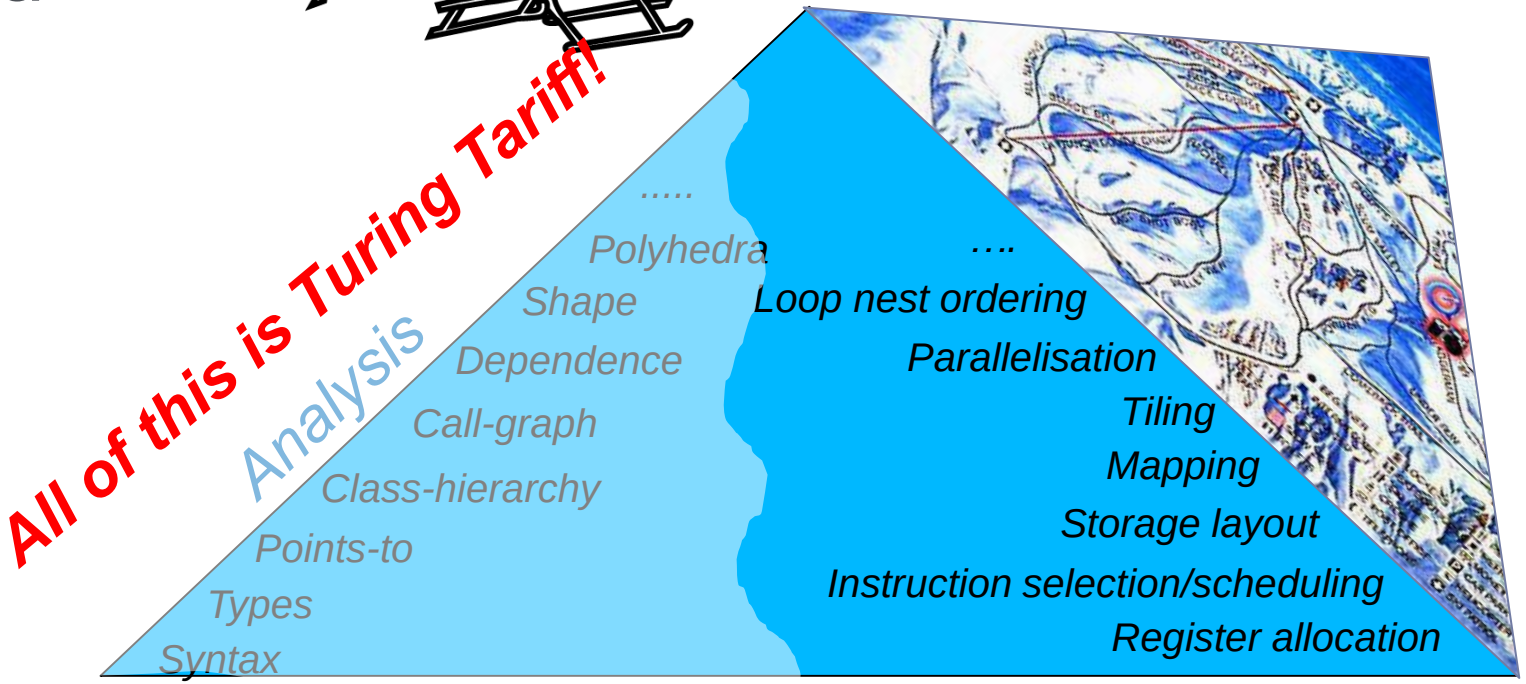
- Analysis is not always the interesting part....
- It's more fun the higher you start!

http://www.nikkiemcdade.com/subFiles/2DExamples.html

- General-purpose programming languages make you pay Turing tariffs!
- **The real art of domain-specific compiler construction is compiler architecture: the design of the representations that make hard problems easy**



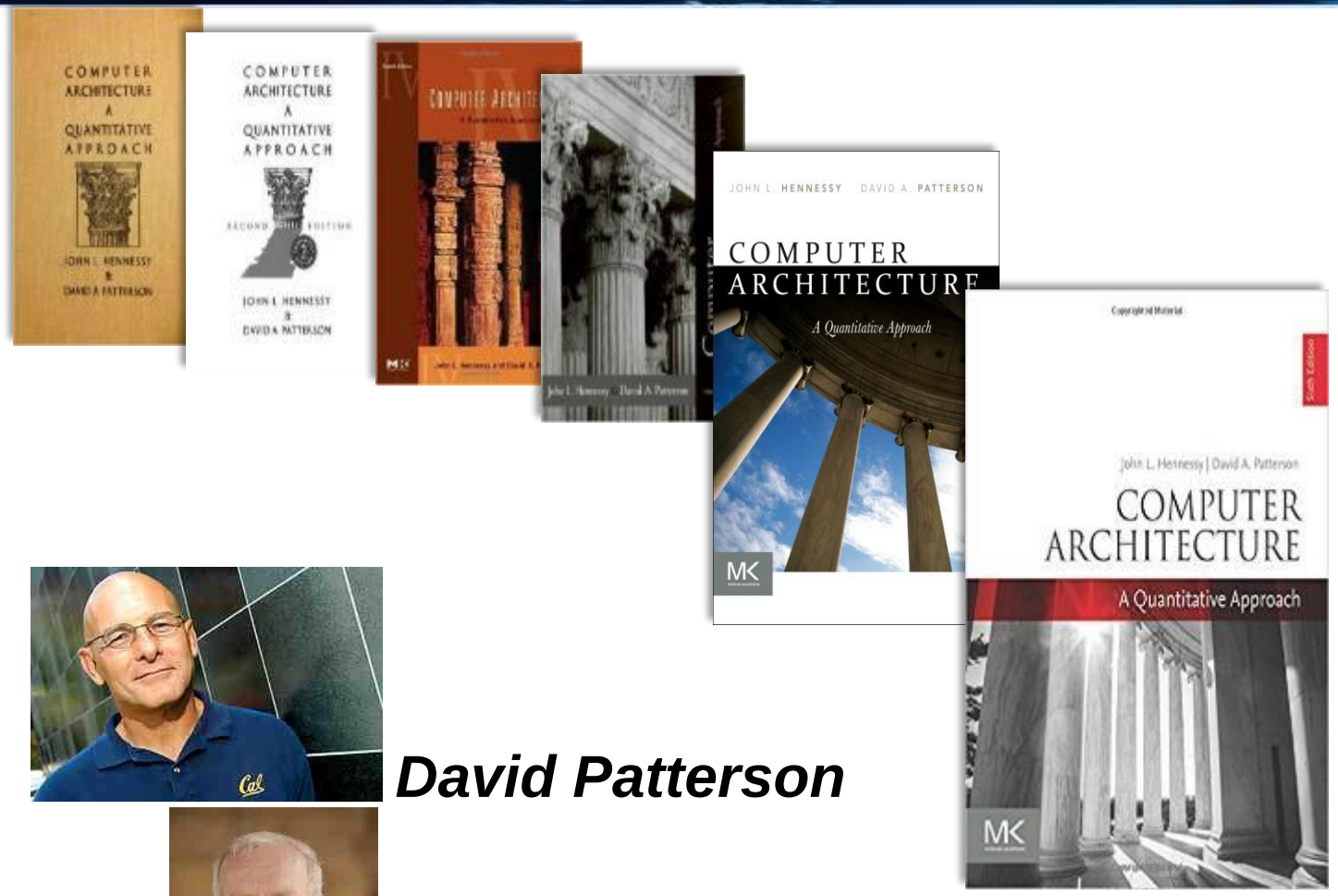
- Compilation is like skiing



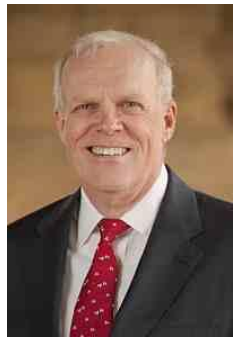
- Analysis is not always the interesting part....
- It's more fun the higher you start!

http://www.nikkiemcdade.com/subFiles/2DExamples.html

Computer architecture – the book



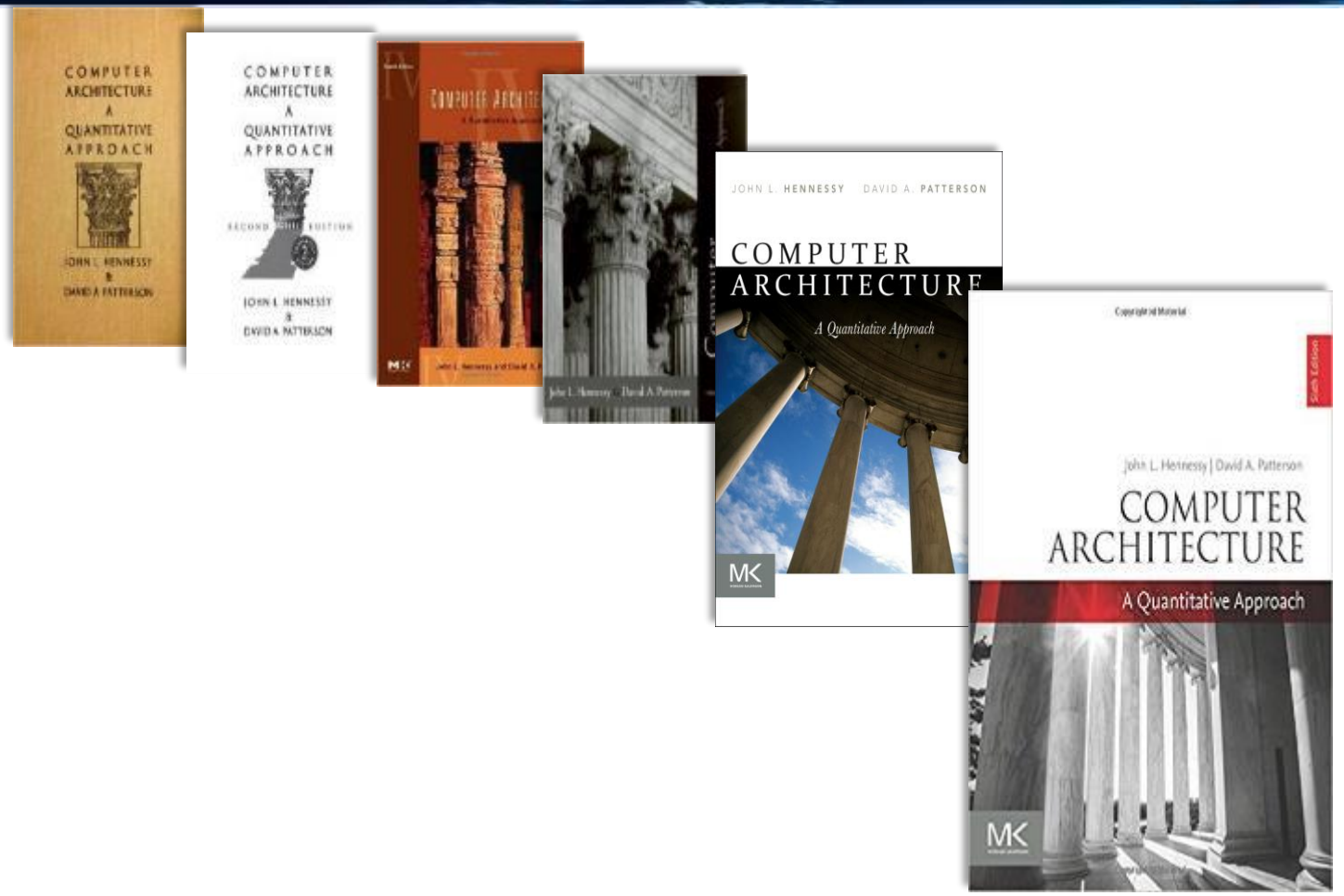
David Patterson

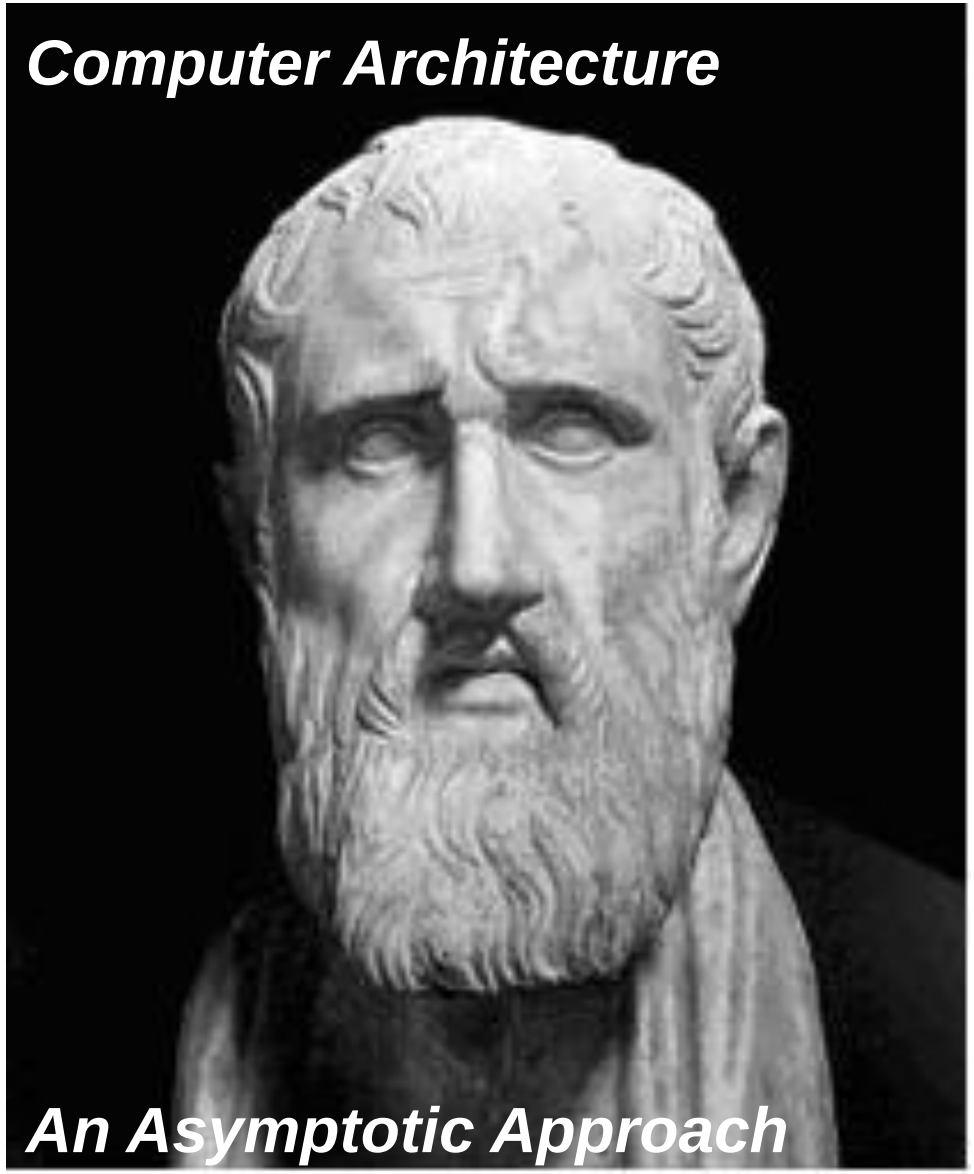


John Hennessy

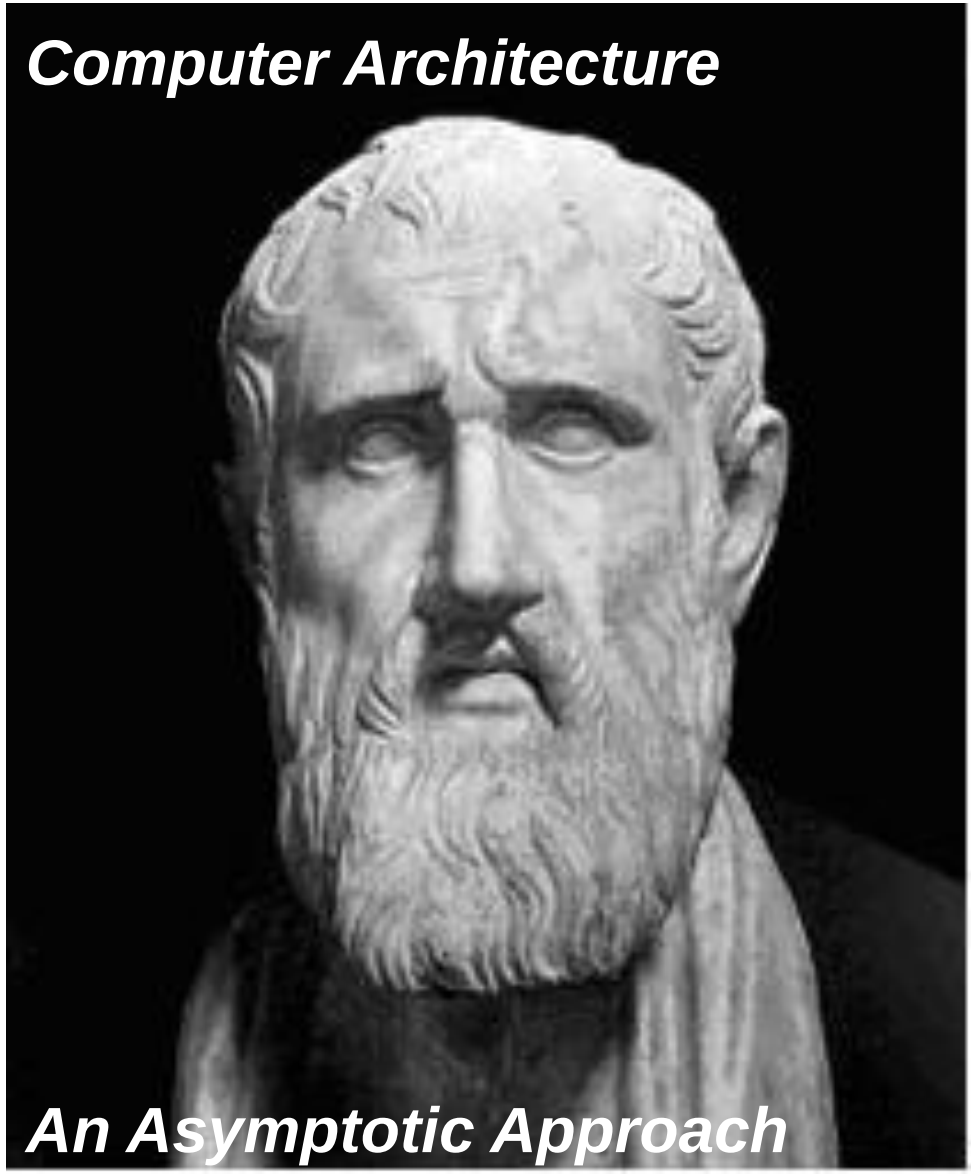
- Computer Architecture: A Quantitative Approach
- Six editions since 1990
- Revolutionary landmark book brought experimental discipline to processor design
- Almost entirely devoid of theory

Computer architecture – the future?





Computer architecture – the future?



- A manifesto
- For computer architecture at the end of Moore's Law
- Where we confront fundamental physical constraints
- Where we have to account for fundamental costs
- Where architectural efficiency is paramount

Conclusions - propositions

A:
- Parallelism is (usually) easy – locality is hard

B:
- Don’t spend your whole holiday carrying your skis uphill

C:
- Domain-specific compiler architecture is not about analysis! It is all about designing representations, and doing the right thing at the right level

D:
- When there’s no more room at the bottom, all efficient computers will be domain-specific

E:
- Design of efficient algorithms will be about designing efficient domain-specific architectures

F:
- All compilers will have a place-and-route phase

Acknowledgements

Partly funded by

- 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 Grant (EP/I006761/1)
- EPSRC “Custom Computing” Platform Grant (EP/I012036/1)
- AMD, Codeplay, Maxeler Technologies