“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 Mudailege, 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 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-wraper
Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM). Firedrake uses sophisticated code generation techniques to enable mathematicians, scientists, and engineers with a very high productivity way to work with state of the art, sophisticated high performance simulations.

Features:

- Expressive specification of any PDE using the Unified Form Language (UFL).
- 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 tensor product elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.
Firedrake is used in:

- **Thetis**: unstructured grid coastal modelling framework

The Thetis project

Thetis is an unstructured grid coastal ocean model built using the firedrake finite element framework. Currently Thetis consists of 2D depth averaged and full 3D baroclinic models. Some example animations are shown below. More animations can be found in the Youtube channel.

- Idealized river plume simulation
- Baroclinic eddies test case
- Thetis Tidal Barrage simulation
- Thetis Two Lagoon Simulation

What is it used for? By whom?

Thetis source code is hosted on Github and is being continually tested using Jenkins.
• 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


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

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.

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.

What is it used for? By whom?

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

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

- **UFL** specifies the (weak form of the) partial differential equation and how it is to be discretised
- Compiler generates PyOP2 kernels and access descriptors

- **GEM**: abstract representation supports efficient flop-reduction optimisations

- **PyOP2**: stencil DSL for *unstructured-mesh*
  - Explicit *access descriptors* characterise access footprint of kernels

- **Loo.py**: vectorization etc

---

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 *
g = Grid(shape=(nx, ny))
u = TimeFunction(name='u', grid=g, 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/
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.
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
Cf Moore’s Law: “circuit density doubles every 18 months”

60 years = 40 × 18 months

So Moore’s Law would predict $2^{40} = 10^{12}$ increase
Algorithmic complexity and scheduling

- We teach that access to a hash table is $O(1)$, i.e. 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* $O(1)$.

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 $O(1)$, i.e., 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.

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 $\sim O(n^5)$

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

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
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.
Suppose there were no more room at the bottom.

How should that change how we think?

About architecture?
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.
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?
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

Suppose there were no more room at the bottom

How should that change how we think?

About architecture?
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

- 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

Suppose there were no more room at the bottom

How should that change how we think?

About architecture?
Suppose there were no more room at the bottom

How should that change how we think?

About architecture?

How architects avoid Turing tariffs

- 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

Suppose there were no more room at the bottom.

How should that change how we think?

About architecture?
How *compilers* avoid Turing tariffs

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?**
Analysis is not always the interesting part....

It’s more fun the higher you start!

Compilation is like skiing

Suppose there were no more room at the bottom

How should that change how we think?

About compilers?
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!
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

All of this is Turing Tariff

- Syntax
- Points-to
- Types
- Class-hierarchy
- Call-graph
- Dependence
- Polyhedra
- Shape
- Loop nest ordering
  - Parallelisation
  - Tiling
  - Mapping
  - Storage layout
  - Instruction selection/scheduling
  - Register allocation

Analysis

- Analysis is not always the interesting part....
- It’s more fun the higher you start!
Computer architecture – the book

- 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?

- 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
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