FEniCSx: Design of the next generation FEniCS libraries for finite element methods

Garth N. Wells

FEM@LLNL

8th November 2022

Part I: Design and development of FEniCSx libraries

Part II: High-performance finite element kernels

Design and development of FEniCSx libraries

Garth Wells, Jorgen Dokken, Michal Habera, Jack Hale, Chris Richardson, Marie Rognes, Matthew Scroggs, Nathan Sime and the FEniCS Project contributors

https://fenicsproject.org

Quiz 1: what does this function do?

```
SUBROUTINE XXXXXX(N,DX,INCX,DY,INCY,C,S)
DOUBLE PRECISION C,S
INTEGER INCX,INCY,N
DOUBLE PRECISION DX(*),DY(*)
DOUBLE PRECISION DTEMP
INTEGER I,IX,IY

IF (n.LE.0) RETURN
IF (incx.EQ.1 .AND. incy.EQ.1) THEN
DO i = 1,n
      dtemp = c*dx(i) + s*dy(i)
      dy(i) = c*dy(i) - s*dx(i)
      dx(i) = dtemp
   END DO
ELSE
   ix = 1
   iy = 1
   IF (incx.LT.0) ix = (-n+1)*incx + 1
   IF (incy.LT.0) iy = (-n+1)*incy + 1
   DO i = 1,n
      dtemp = c*dx(ix) + s*dy(iy)
      dy(iy) = c*dy(iy) - s*dx(ix)
      dx(ix) = dtemp
      ix = ix + incx
      iy = iy + incy
   END DO
END IF
RETURN
END
```

Quiz 1: answer

Plane rotation from reference BLAS (`DROT`)

Quiz 2: What does this mean?

$$-\nabla^2 u = f \quad \text{in } \Omega \subset \mathbb{R}^3$$

$$u = 0 \quad \text{on } \partial\Omega$$

Overview

- A domain-specific language (DSL) for variational forms
- Legacy FEniCS 1.0
- What has worked, and hasn’t
- FEniCSx design and development

Poisson equation

Find $u \in H_0^1(\Omega)$ such that

$$a(u, v) = L(v) \quad \forall v \in H_0^1(\Omega)$$

where

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v \, dx$$

$$L(v) := \int_\Omega f v \, dx$$

Poisson equation in a DSL (UFL)

```
geometry = VectorElement("Lagrange", triangle, 2)
mesh = Mesh(geometry)
element = FiniteElement("Lagrange", triangle, 3)
V = FunctionSpace(mesh, element)

u, v = TrialFunction(V), TestFunction(V)
f = Coefficient(V)
a = inner(grad(u), grad(v)) * dx
L = inner(f, v) * dx
```

Hyperelasticity (1)

```
element = VectorElement("Lagrange", tetrahedron, 1)
du = TrialFunction(element)      # Incremental displacement
v  = TestFunction(element)       # Test function

u = Coefficient(element)         # Displacement from previous
                                 # iteration

# Kinematics
D, I = len(u), Identity(d)
F = I + grad(u)                  # Deformation gradient
C = F.T * F                      # Right Cauchy-Green tensor

# Invariants of deformation tensors
Ic, J = tr(C), det(F)

E, nu  = 10.0, 0.3
mu, lmbda = E/(2*(1 + nu)), E*nu/((1 + nu)*(1 - 2*nu))
```

Hyperelasticity (2)

```
# Stored strain energy density (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2

# Total potential energy
Pi = psi*dx

# First variation of Pi (directional derivative about u in
# the direction of v). Newton solver will drive this to zero
F = derivative(Pi, u, v)

# Compute Jacobian of F. Matrix operator in Newton's method
J = derivative(F, u, du)
```

A Poisson solver (FEniCS 1.0)

```
. . .
u, v = TrialFunction(V), TestFunction(V)
f = Expression("10*exp(-(pow(x[0] - 0.5, 2) \
    + pow(x[1] - 0.5, 2)) / 0.02)", degree=2)
g = Expression("sin(5*x[0])", degree=2)
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*ds

u = Function(V)
solve(a == L, u, bc)
```

Unified Form Language (UFL)

- A domain-specific embedded language for variational forms
- Embedded in Python
- Highly expressive
- Implements various form manipulations
- Generates abstract representation (DAG) of variational problems
- Requires a backend to generate concrete code

Alnæs, Logg, Ølgaard, Rognes, Wells (2014), doi:10.1145/2566630

Distinguishing technologies for FEniCS 1.0: domain specific languages and code generation

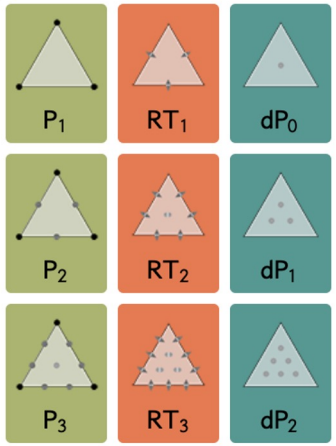
- FInite Element Automatic Tabulator (FIAT) 2002-
- Unified Form Language (UFL), 2008 -
- FEniCS Form Compiler (FFC), 2004 -

UFL, FE symbolic language

inner(grad(**u**), grad(**v**)) * dx

FFC, UFL->C compiler

```
void tabulate_tensor_poisson_cell_integral_43165b9e21c850b7e46d14f843b
                                        const ufc_scalar_t* c,
                                        const double* restrict coordinate_
                                        const int* unused_local_index,
                                        const int* cell_orientation)
{
    // Precomputed values of basis functions and precomputations
    // FE* dimensions: [entities][points][dofs]
    // PI* dimensions: [entities][dofs][dofs] or [entities][dofs]
    // PM* dimensions: [entities][dofs][dofs]
    alignas(32) static const ufc_scalar_t FE3_C0_D01_Q1[1][1][2] = { {
    // Unstructured piecewise computations
    const double J_c0 = coordinate_dofs[0] * FE3_C0_D01_Q1[0][0][0] +
    const double J_c3 = coordinate_dofs[1] * FE3_C0_D01_Q1[0][0][0] +
    const double J_c1 = coordinate_dofs[0] * FE3_C0_D01_Q1[0][0][0] +
    const double J_c2 = coordinate_dofs[1] * FE3_C0_D01_Q1[0][0][0] +
```



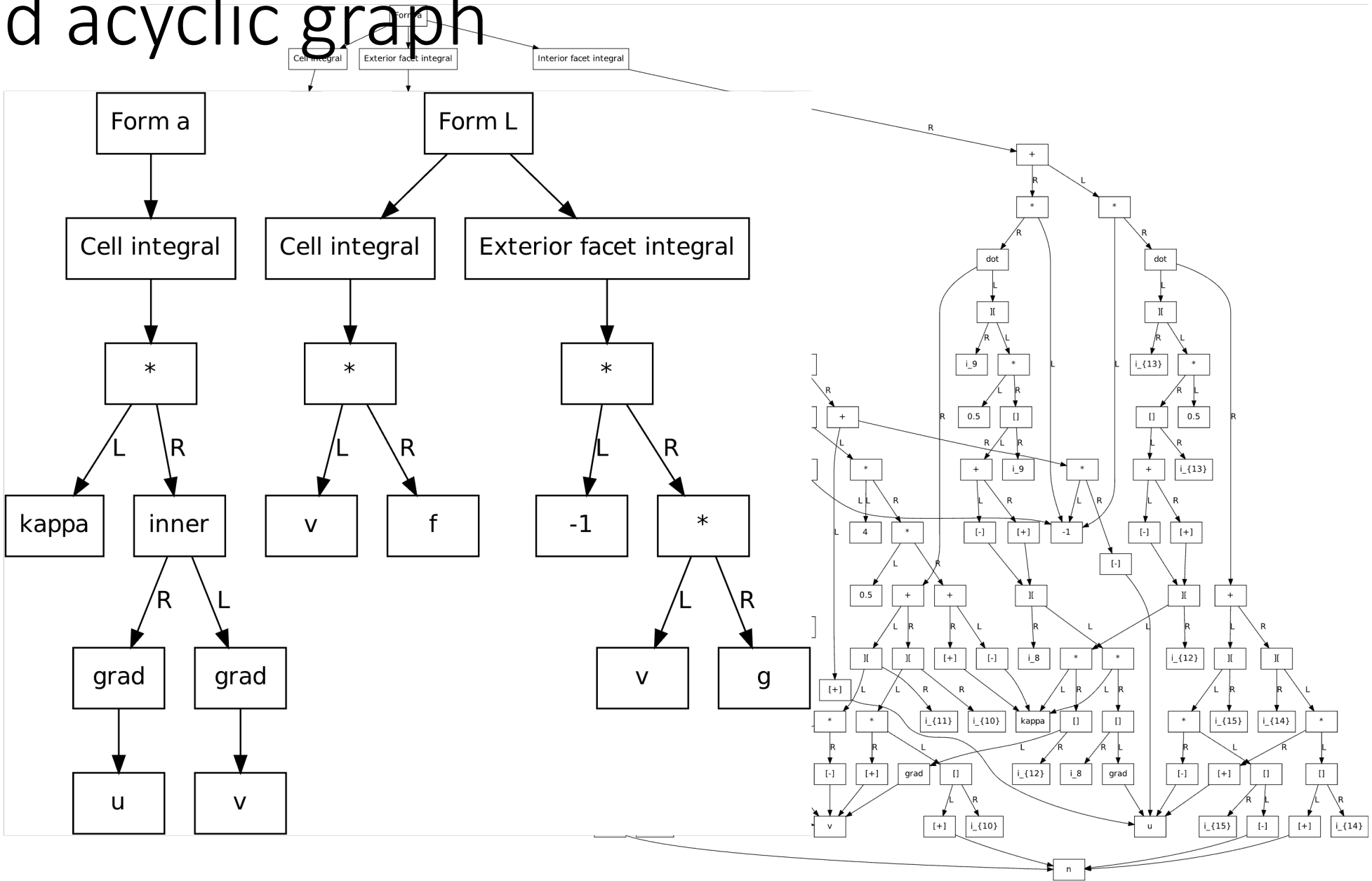
FIAT, FE oracle

u, v in (Lagrange, 1st order)

Weighted Poisson equation in UFL

```
V = FiniteElement("Lagrange", triangle, 1)

u = TrialFunction(V)
v = TestFunction(V)

k = Coefficient(V)
f = Coefficient(V)
g = Coefficient(V)

a = k*inner(grad(u), grad(v))*dx
L = inner(f, v)*dx - inner(g, v)*ds
```

Directed acyclic graph



Mathematical intent vs algorithm vs implementation

- UFL expresses and preserves mathematical intent
- It does not encode algorithmic details
- It does not encode implementation details, e.g. assembly, strategy, linear solver, target architecture/system, etc

FEniCS Form Compiler (FFC)

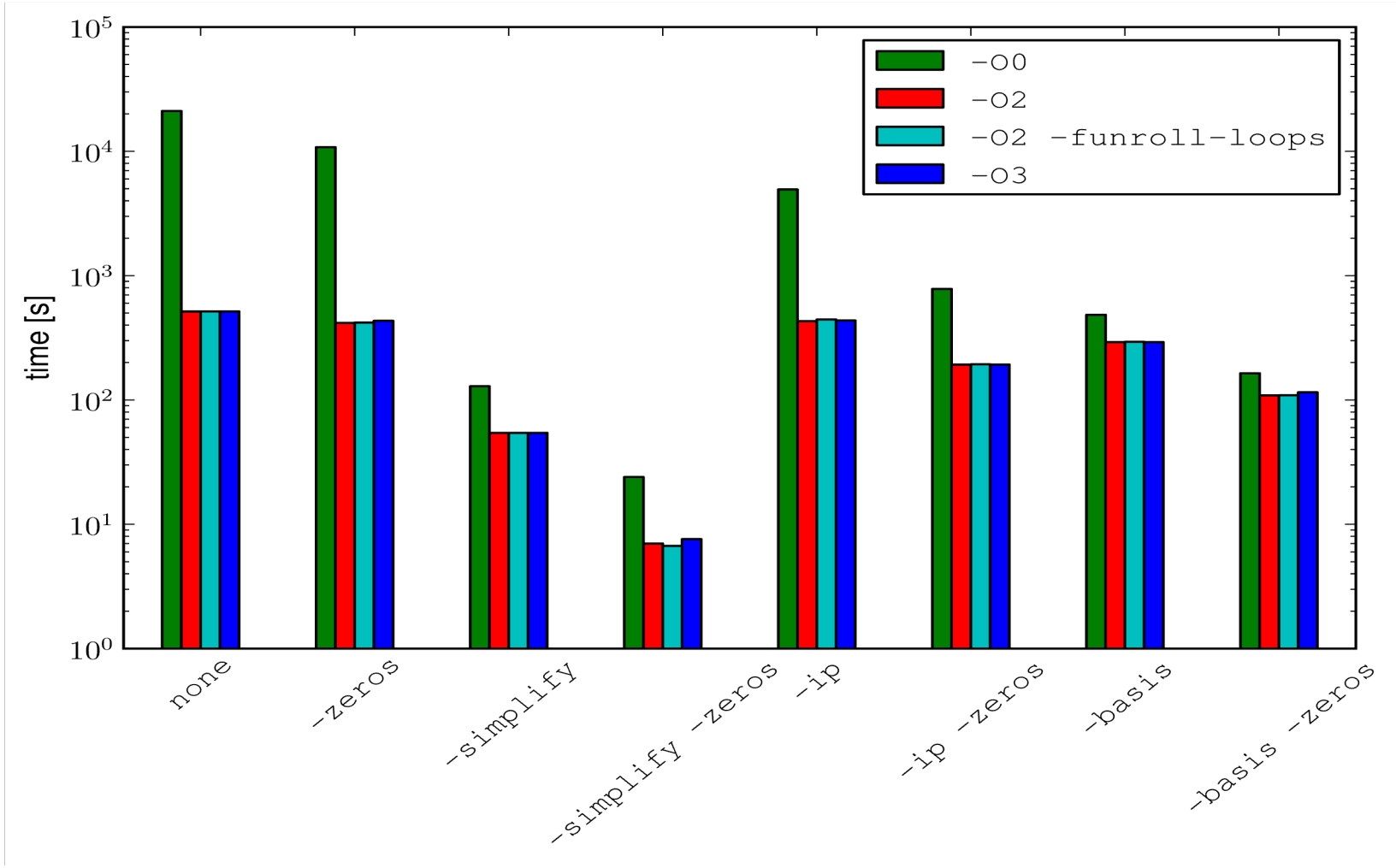
Code generator (FFC) takes UFL abstract representation and generates code in target language

Kirby, Logg (2005) ACM TOMS; Logg, Ølgaard, Rognes, Wells (2012)

Generated code

```
void tabulate_tensor_integral_cell_otherwise_17e5(ufc_scalar_t* restrict A, const
ufc_scalar_t* restrict w, const ufc_scalar_t* restrict c, const double* restrict
coordinate_dofs, const int* restrict unused_local_index,  const uint8_t* restrict
quadrature_permutation, const uint32_t cell_permutation)

{

    alignas(32) static const double weights_39d[6] = { 0.054975871827661,
0.054975871827661, 0

   // Precomputed values of basis functions and precomputations

    // FE* dimensions: [permutation][entities][points][dofs]

    alignas(32) static const double FE4_C0_D01_Q39d[1][1][6][10] =

        { { { { -0.2890278173026988, 0.0, 0.2890278173026942, 5.331925347622843, -
1.656111269210768, -0.1132137388906001, 0.1132137388906256, 1.656111269210752, -
5.331925347622805, 0.0 },

              { -0.2890278173026957, 0.0, 2.656111269210756, -0.2988792219033607,
1.607609848407356, 1.357232047307428, -3.724315499215482, 0.1856654830127168,
0.2988792219033352, -1.79327533142008 } } } } };

    alignas(32) static const double FE8_C0_D01_Q39d[1][1][1][3] = { { { { -1.0, 0.0, 1.0 }
} } };

    alignas(32) static const double FE8_C0_D10_Q39d[1][1][1][2] = { { { { -1.0, 1.0 } } }
};

    for (int iq = 0; iq < 6; ++iq)

. . . .
```

Compiler optimisations and representations circa 2009



Ølgaard & Wells (2010), ACM TOMS

DOLFIN: problem solving environment

- DOLFIN is the FEniCS Project 1.0 problem solving environment
- Synthesises domain-specific language, code generation, linear algebra, domain representation (mesh), . . .
- C++ and Python interfaces
- Design reflects mathematical abstractions
- Manages parallel aspects of a simulation

Logg and Wells (2010), ACM TOMS

Issues and criticisms of FEniCS 1.0 libraries

Works well from user-perspective if *remaining* within the supported abstractions

Design limitations

- Hard to break out from supported abstractions
- Difficult to extend/build upon, especially from Python interface

Consequences and implementation issues

- Difficult/impossible to experiment with new methods, especially at a low level
- Mixture of mature and immature/niche technologies in core library
- Inconsistent behaviour in parallel
- Slowed development progress

Performance and other issues

- Performance had slipped
- Hard to ‘see through’ the code to understand performance
- Too much unnecessary code generation and JIT (complexity, slow pre-compilation, hard to extend)
- Easy for users to write slow code

- More than one way to do the same thing without good reason
- Too implicit, excessive caching of objects, hidden expensive steps in the interests of ‘expressiveness’
- Single type only support

Start again . . . (not quite): FEniCS-X

Keep the demonstrated strengths with high level abstractions

Allow all operations to be computed/implemented ‘manually’

Highly efficient implementations implemented at a high level

Consistent parallel behaviour

Hardware-friendly

The supporting tools have all changed

Remarkable progress in supporting software tools since early FEniCS developments in mid-2000s

Example 1: NumPy didn't exist

`numeric` and `numarray`

Example 2: C++/Python interfacing

`SWIG`: Automated wrapping of C++ interface to Python.

We had 18k lines of 'SWIG language' code to guide the automation

`pybind11`: Manual wrapping of C++ interface

1,500 lines of C++, faster and more flexible

Example 3: C++ complexity

Object oriented designs and templating became gratuitous in 2000s

New tools became available...

Numba
Python and NumPy/LLVM
JIT compiler

pybind11
C++14/Python bindings

C++20

SYCL

Auto-vectorization

```
remark: vectorized loop (vectorization width: 4,
                         interleaved count: 4)
```

Unified Form Language (UFL)

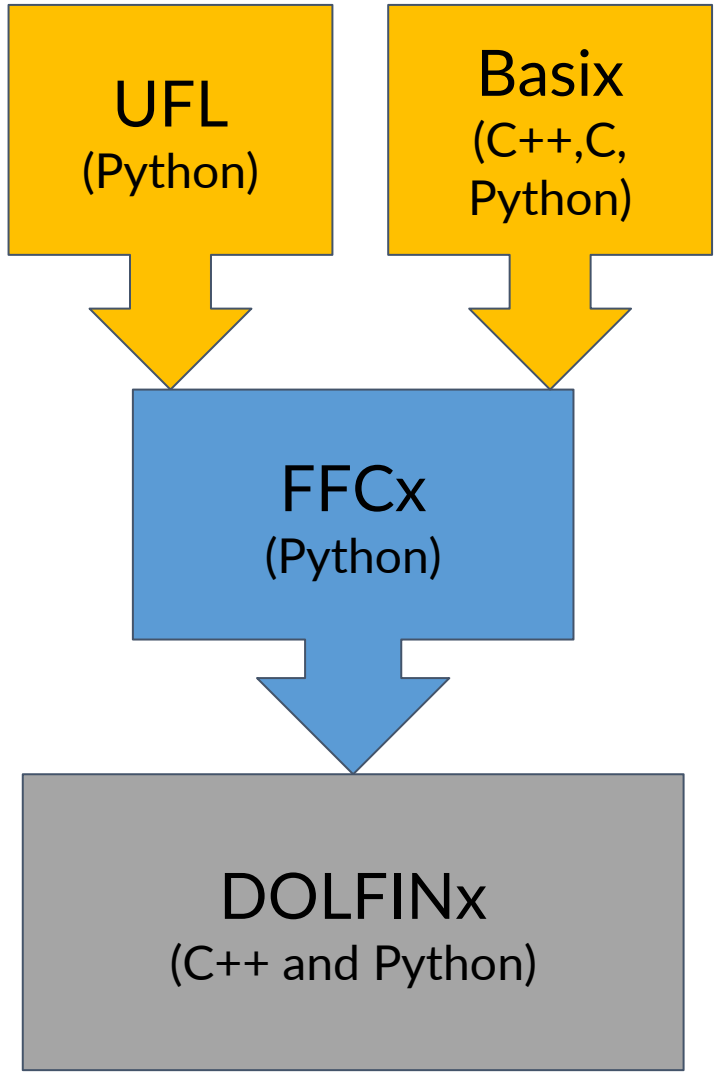
Largely unchanged, some extensions

A revision underway

FEniCSx: modular

Python

Abstract

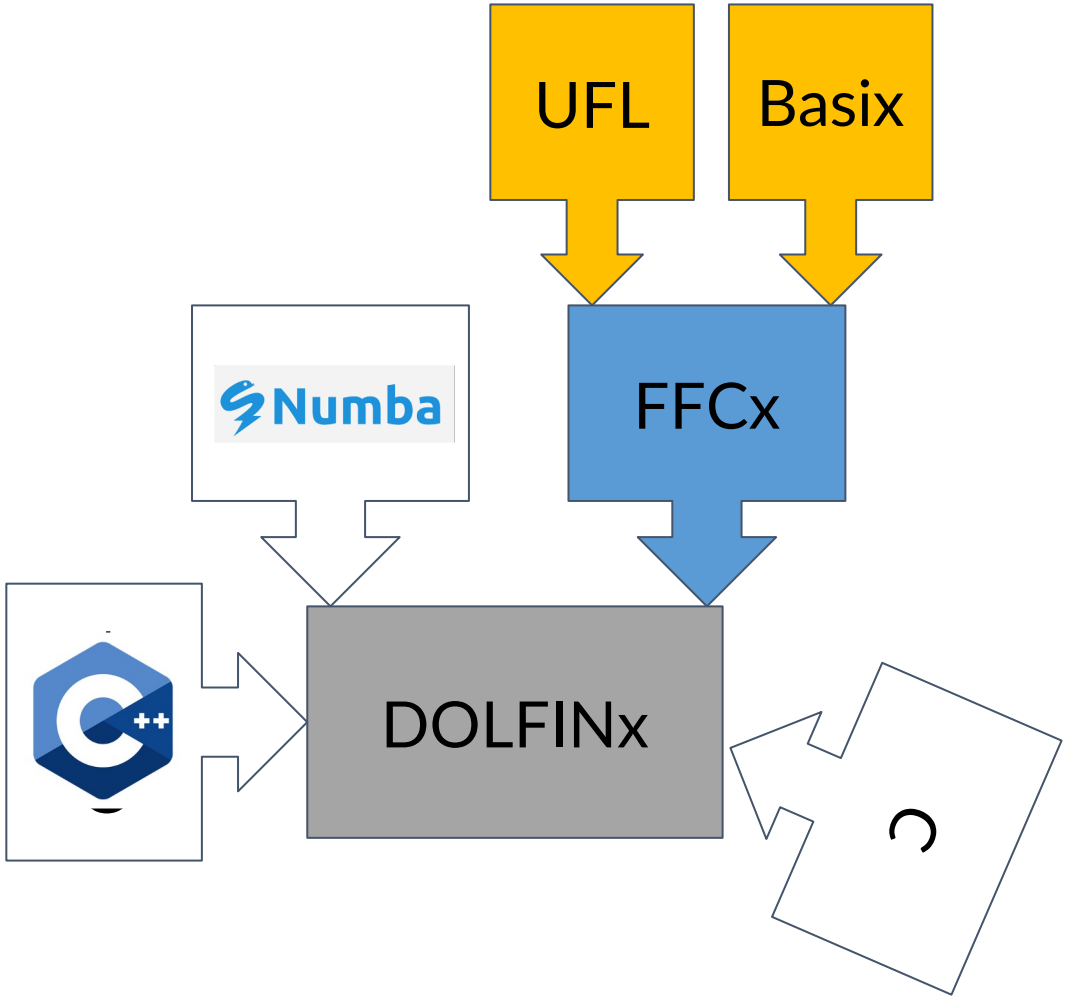


Generates kernels (C code)

Transform

Orchestrate/execute

FEniCSx: extensible



FEniCSx is designed to be extensible

- Use automated tools when suitable
- Allows fully custom implementations, e.g.:
  - Finite element kernels
  - Assemblers
  - Input/output
  - Linear algebra backends

Solver interface DOLFINx

- Functional and data-centric design
- Pure functions - stateless
- Less object-oriented design and less data encapsulation
- More explicit behaviour
- Everything should be possible ‘by hand’

Package (without deps)	C++/C lines	Python lines
DOLFIN	90,000	22,000
DOLFINx	**28,000**	**12,269**
deal.ii	830,000	
PETSc	580,000	
Firedrake		37,000
Eigen	125,000	

generated SLOCCount

Working on data: expression evaluation

Old (uses JIT of C strings)

```
f = Expression("exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)))",
               degree=2)
```

New (NumPy-based)

```
def f(x):
    return np.exp(-(x[0]-0.5)**2) + (x[1]-0.5)**2
f0,f1,f2 = Function(V0), Function(V1), Function(V1)
f0.interpolate(f)
f1.interpolate(f)
f2.interpolate(lambda x: np.exp(-(x[0]-0.5)**2) + (x[1]-0.5)**2)
```

(not) Working on data: expression evaluation

```
class Source : public Expression
{
  void eval(Array<double>& values, const Array<double>& x) const
  {
    double dx = x[0] - 0.5;
    double dy = x[1] - 0.5;
    values[0] = 10*exp(-(dx*dx + dy*dy) / 0.02);
  }
};
auto f = std::make_shared<Source>();
L.f = f;
```

Working on data: pure functions & expression evaluation

```
Function f(V);
f.interpolate(
      [](auto x) -> std::pair<std::vector<T>, std::vector<std::size_t>>
      {
        std::vector<T> f(x.extent(1));
        for (std::size_t p = 0; p < x.extent(1); ++p)
          f[p] = std::sin(2 * std::numbers::pi * x(0, p));
        return {f, {f.size()}};
      });

Function g(W);
g.interpolate(f)
```

Functions and data

```
facets = locate_entities(msh, dim=1,
                         marker=lambda x: np.logical_or(np.isclose(x[0], 0.0),
                                                        np.isclose(x[0], 2.0)))
```

Supporting linear algebra backends

Old approach

- Nightmare of class hierarchies and boilerplate
- Attempts to shoehorn different backends into common interfaces

New approach

- Functional with captures, no classes
- Trivial to support new backends without modifying the library

Functional approach: assembly into linear algebra backends

PETSc

```
// Matrix insertion function, captures PETSc matrix pointer
auto mat_add = [A](std::span<const std::int32_t> rows,
                   std::span<const std::int32_t> cols,
                   std::span<const PetscScalar> vals) -> int
    {
      PetscErrorCode ierr;
      ierr = MatSetValuesLocal(A, rows.size(), rows.data(), cols.size(),
                               cols.data(), vals.data(), ADD_VALUES);
      return ierr;
    };

// Assemble bilinear form into a matrix
template <typename T>
void assemble_matrix(auto mat_add, const Form<T>& a);
```

Functional approach: assembly into linear algebra backends

Tpetra

```
// Matrix insertion function, captures Tpetra matrix reference
auto mat_add = [&A](auto rows, auto cols, auto vals) -> int
    {
      . . .
      A->sumIntoLocalValues(. . .);
      return 0;
    };

// Assemble bilinear form into a matrix
template <typename T>
void assemble_matrix(auto mat_add, const Form<T>& a);
```

Functional approach: custom mesh partitioners

```
using CellPartitionFn
    = std::function<graph::AdjacencyList<std::int32_t>(
        MPI_Comm comm, int nparts, int tdim,
        const graph::AdjacencyList<std::int64_t>& cells)>;

// Create a mesh using a provided parallel partitioning function
Mesh create_mesh(MPI_Comm comm, const
                 graph::AdjacencyList<std::int64_t>& cells,
                 const fem::CoordinateElement& element,
                 std::span<const double> x,
                 std::array<std::size_t, 2> xshape,
                 CellPartitionFn partitioner);
```

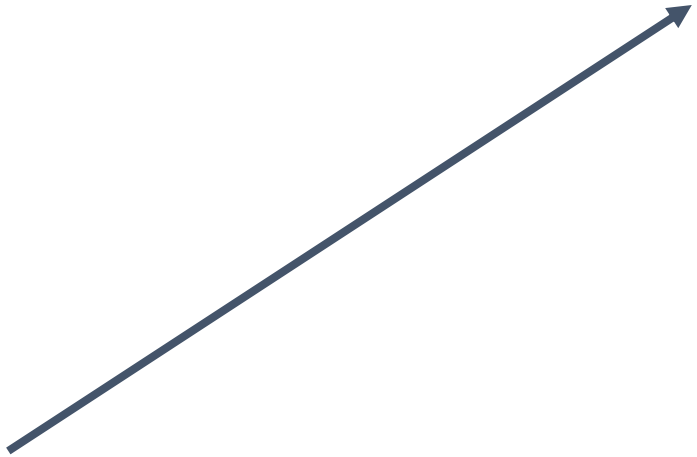
Example: JIT kernel and built-in assembler, Stokes flow

```
P2 = ufl.VectorElement("Lagrange", msh.ufl_cell(), 2)
P1 = ufl.FiniteElement("Lagrange", msh.ufl_cell(), 1)
V, Q = FunctionSpace(msh, P2), FunctionSpace(msh, P1)

(u, p) = ufl.TrialFunction(V), ufl.TrialFunction(Q)
(v, q) = ufl.TestFunction(V), ufl.TestFunction(Q)

a = form([[inner(grad(u), grad(v)) * dx, inner(p, div(v)) * dx],
          [inner(div(u), q) * dx,         None]])

# Assemble into a block-nested matrix
A = fem.petsc.assemble_matrix_nest(a, bcs=bcs)
A.assemble()
```

Example: static condensation kernel (1)

```
@numba.cfunc(c_signature, nopython=True)
def kernel(A_, w_, c_, coords_, e, c):
    A = numba.carray(A_, (Usize, Usize))
    A00 = numpy.zeros((Ssize, Ssize))
    kernel00(ffi.from_buffer(A00), ...)
    A01 = numpy.zeros((Ssize, Usize))
    kernel01(ffi.from_buffer(A01), ...)
    A10 = numpy.zeros((Usize, Ssize))
    kernel10(ffi.from_buffer(A10), ...)

  # A = - A10 * A00^{-1} * A01
  A[:, :] = - A10 @ numpy.linalg.solve(A00, A01)
```

NumPy supported operations, many implemented with BLAS, LAPACK



Example: static condensation assemble

```
a = Form([U, U])
a.set_tabulate_tensor(..., knl.address)

A = assemble_matrix(a_cond)
A.assemble()
```

Assembly: user Python implementation

```
@numba.njit
def area(x0, x1, x2) -> float:
    """Compute the area of a triangle embedded in 2D
    from the three vertices"""
    a = (x1[0] - x2[0])**2 + (x1[1] - x2[1])**2
    b = (x0[0] - x2[0])**2 + (x0[1] - x2[1])**2
    c = (x0[0] - x1[0])**2 + (x0[1] - x1[1])**2
    return np.sqrt(2*(a*b + a*c + b*c) - (a**2 + b**2 + c**2)) / 4.0
```

```
@numba.njit
def assemble_vector(b, mesh, x, dofmap):
    connections, pos = mesh
    q0, q1 = 1/3.0, 1/3.0
    for i, cell in enumerate(pos[:-1]):
        num_vertices = pos[i + 1] - pos[i]
        c = connections[cell:cell + num_vertices]
        A = area(x[c[0]], x[c[1]], x[c[2]])
        b[dofmap[i * 3 + 0]] += A * (1.0 - q0 - q1)
        b[dofmap[i * 3 + 1]] += A * q0
        b[dofmap[i * 3 + 2]] += A * q1
```

Performance: MPI neighborhood collectives

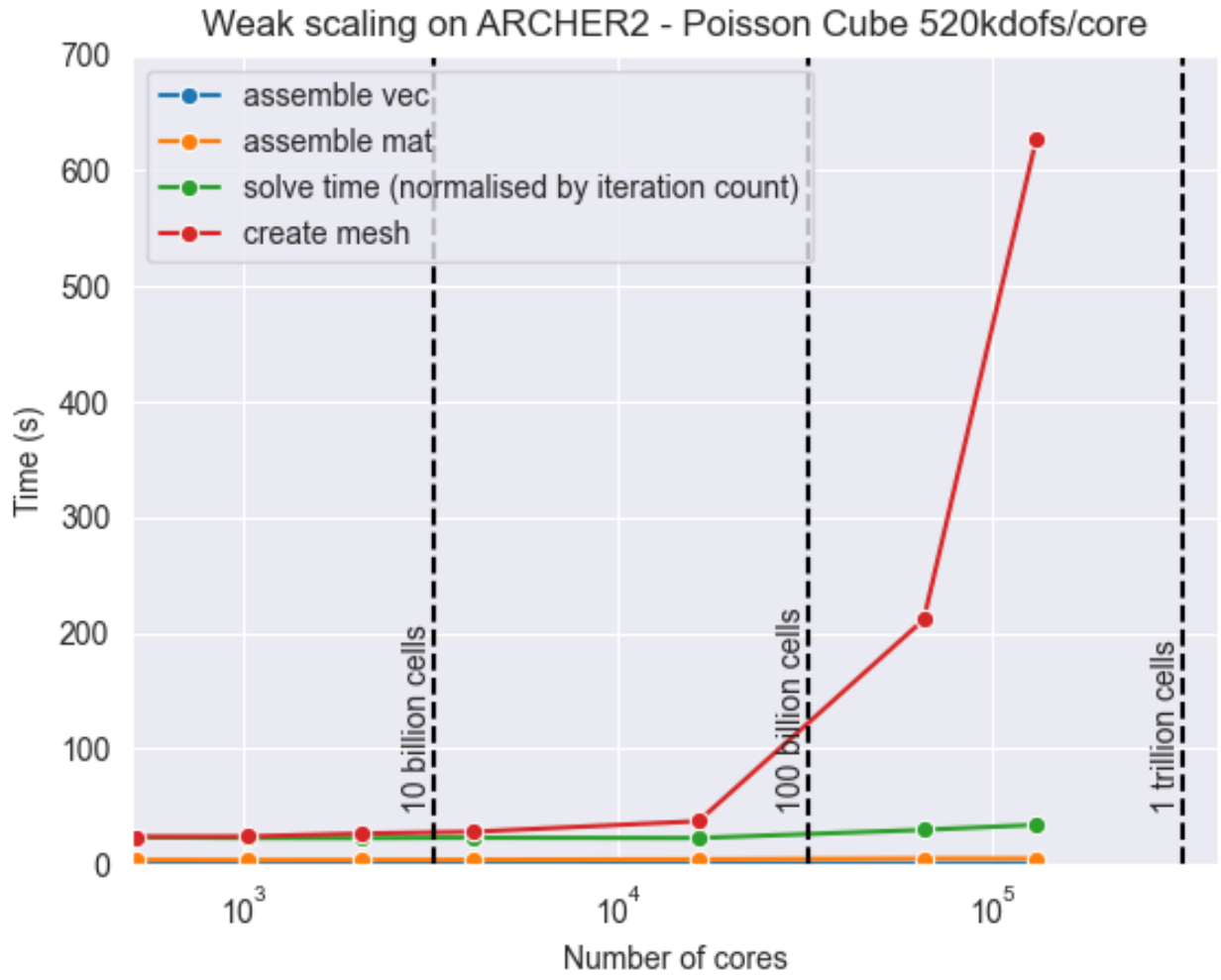
Neighbourhood collectives and unstructured grid methods are a match made in heaven

But, you need to build the neighbourhoods

Mesh creation blowing up in time due to non-scalable MPI all-to-all calls for building neighbourhoods

- 412 billion cells
- 131,072 cores

```
UnitCube (64x64x64) to be refined 6 times
------------------------------------------------------------
Test problem summary
  Problem type:    poisson
  Scaling type:    weak
  Num processes:   131072
  Num cells        412316860416 (412 billion)
  Total degrees of freedom: 68769820673 (68.8 billion)
  Average degrees of freedom per process: 524672
------------------------------------------------------------
```



Neighbourhood building algorithms

We know outgoing neighbors, but need to find incoming neighbors…

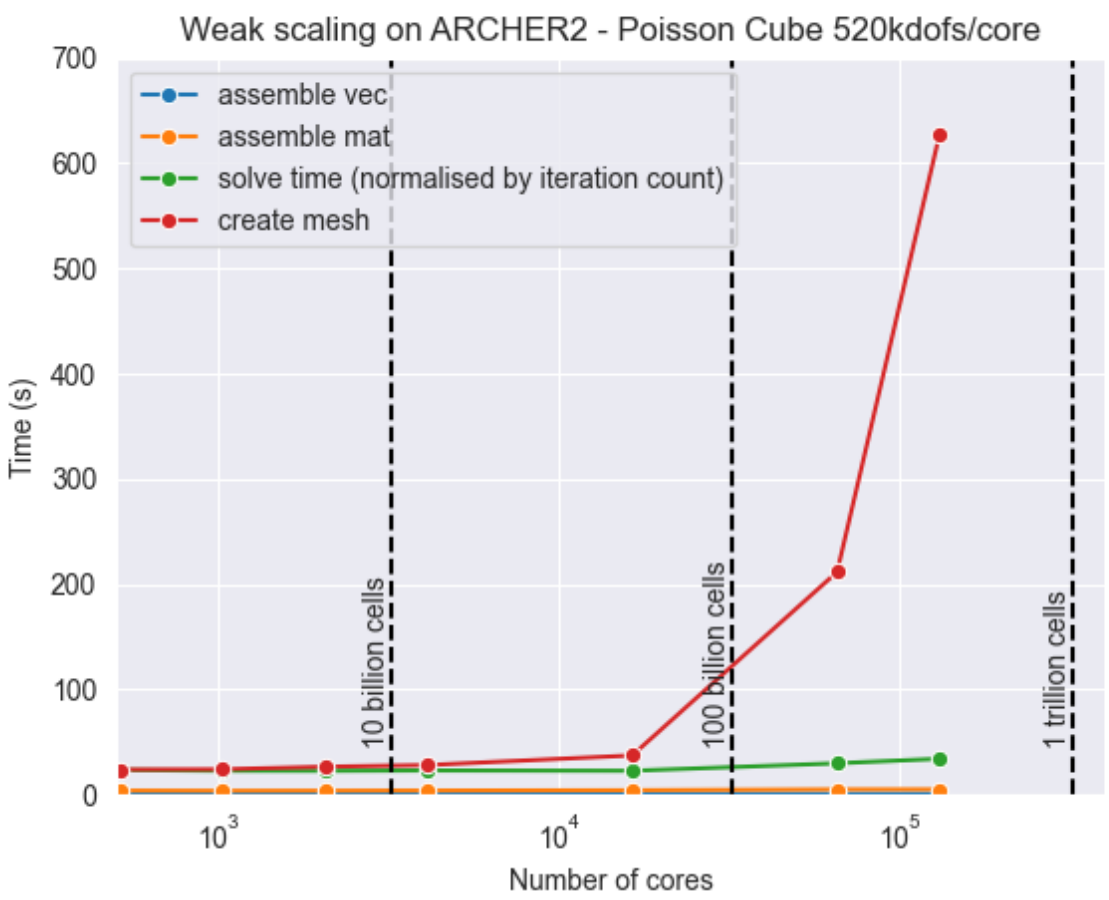
NBX algorithm

Just send small data to neighbor using “synchronous send” (only “completes” once receive acknowledged). MPI-3.

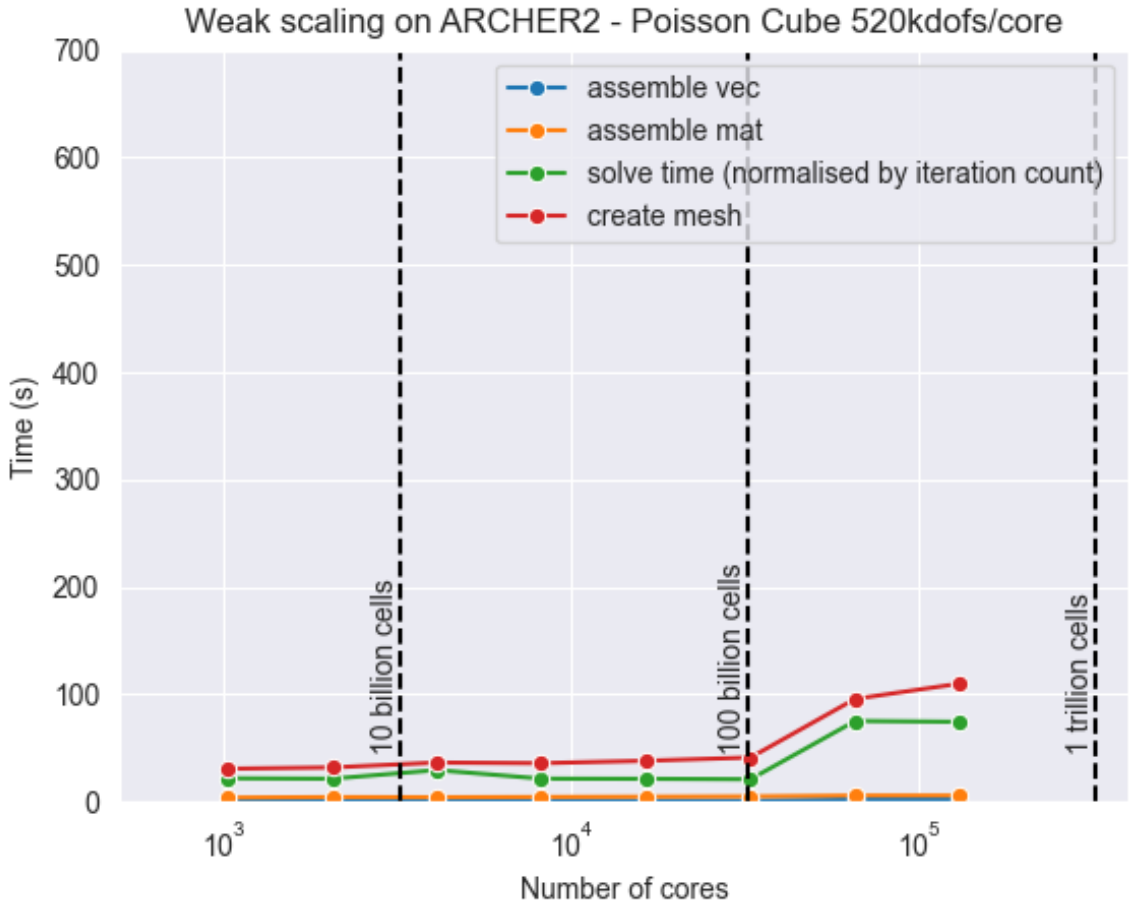
Then use “non-blocking barrier” to wait for all processes to complete.

T. Hoefler, C. Siebert and A. Lumsdaine. ‘Scalable communication protocols for dynamic sparse data exchange’, *ACM Sigplan Notices* 45.5 (2010): 159-168

Using NBX for neighbourhood detection



before



after

FEniCS Form Compiler (FFCx)

Major simplifications

Generates C code (rather C++, GPU code under development)

Now generates minimal ‘canonical data’

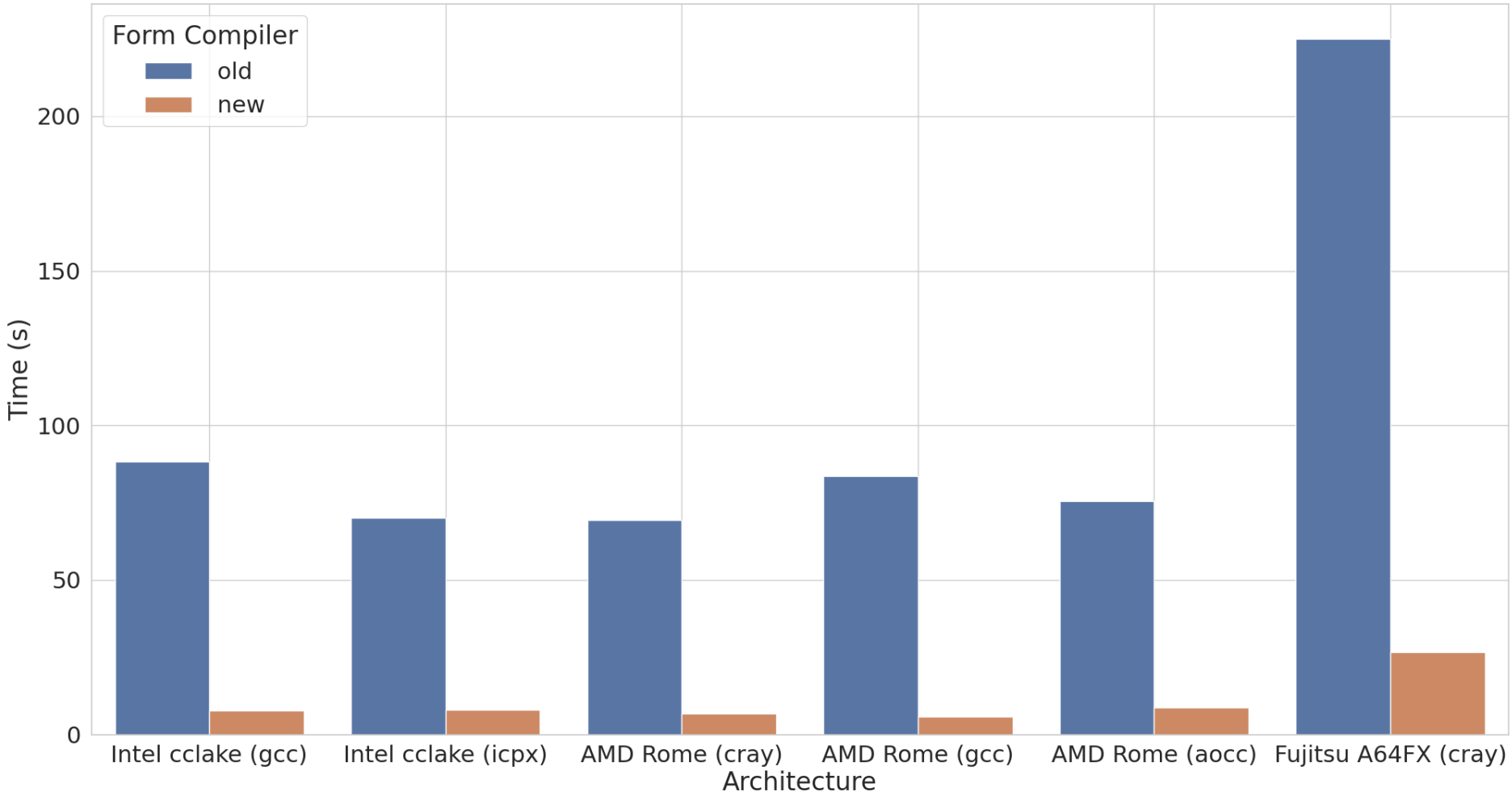
Vectorisation friendly code

Now supports different types, including complex numbers

Different cells types, sum factorisation

Finite element kernels on CPUs

Matrix assembly for the curl-curl operator in 3D using Nedelec 1st kind (degree 4)



FFCx: UFL input to kernel code

UFL input

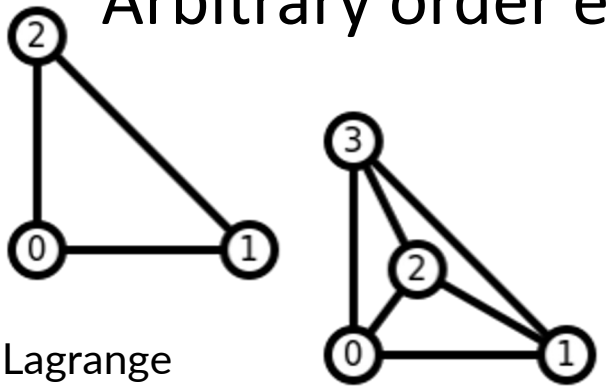
```
coords = VectorElement("P", triangle, 2)
mesh = Mesh(coords)
dx = dx(mesh)
element = FiniteElement("P", mesh.ufl_cell(), 2)
space = FunctionSpace(mesh, element)
u = TrialFunction(space)
v = TestFunction(space)
f = Coefficient(space)
a = inner(grad(u), grad(v)) * dx
```

kernel code

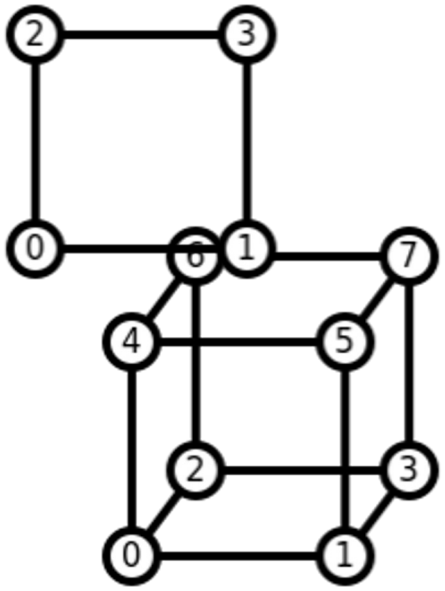
```
void tabulate_tensor_integral_68a(double* restrict A,
const double* restrict w,. . .)
{
  // Quadrature rules
  static const double weights_39d[6] = {
0.054975871827661, 0.054975871827661, … };
  // Precomputed values of basis functions and
precomputations
  // FE* dimensions:
[permutation][entities][points][dofs]
  static const double FE3_C0_D01_Q39d[1][1][6][6] =
    { { { { 0.6336951459609197, . . . },
for (int iq = 0; iq < 6; ++iq)
  {
    // Quadrature loop body setup for quadrature rule
    // Varying computations for quadrature rule 39d
    double J_c0 = 0.0;
. . .
```

Basix: finite element oracle

Tabulate basis functions
Quadrature schemes
Interpolation operators
Arbitrary order elements

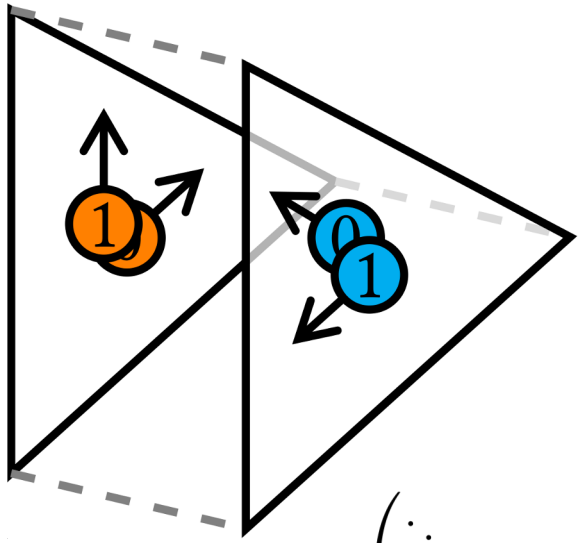


Lagrange
Nédélec (first kind)
Nédélec (second kind)
Raviart–Thomas
Brezzi–Douglas–Marini
Bubble
Crouzeix–Raviart
Regge
Custom



Lagrange (Q)
Nédélec
Raviart–Thomas
Bubble
DPC
Serendipity
Custom

Transformations, esp. for high-order elements, on polyhedral cells. No requirement for meshes to be ordered



$$\begin{pmatrix} \ddots & & \\ & -1 & -1 \\ & 1 & 0 \\ & & \ddots \end{pmatrix}$$

Scroggs, Baratta, Richardson & Wells. Basix: a runtime finite element basis evaluation library, *Journal of Open Source Software* 7(73), 2022, 3982.

Scroggs, Dokken, Richardson & Wells. Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes. ACM TOMS, 2022.

FEniCSx summary

- Data-oriented, functional design (what can it do, not what is it)
  - Transparent performance
  - Easy to reason with in parallel
  - User-injected functions/kernels
  - Supports different languages straightforwardly
  - Lends itself to GPU implementations
- Remarkably small codebase
- C++ and Python interfaces
- No pre-defined operators/kernels
- (minimal) JIT for performant Python for dynamically constructed problems

A finite element library is fundamentally (i) adjacency lists, (ii) algorithms that build and manipulate adjacency lists, and (iii) element kernels

Part II: High-performance finite element kernels

Igor Barratta, Chris Richardson, Garth Wells

Roofline models

Standard

$$R = \min\left\{F_{\max}, \frac{f_d}{b_d} B_{\max}\right\}$$

R maximum compute flops

$F_{\max}$ CPU peak flops

f_d flops per degree-of-freedom

b_d number of transfers to/from main memory

Throughput (dofs/s)

$$T = \alpha \frac{R}{f_d}$$

α efficiency $(0 < \alpha \leq 1)$

Cache-aware

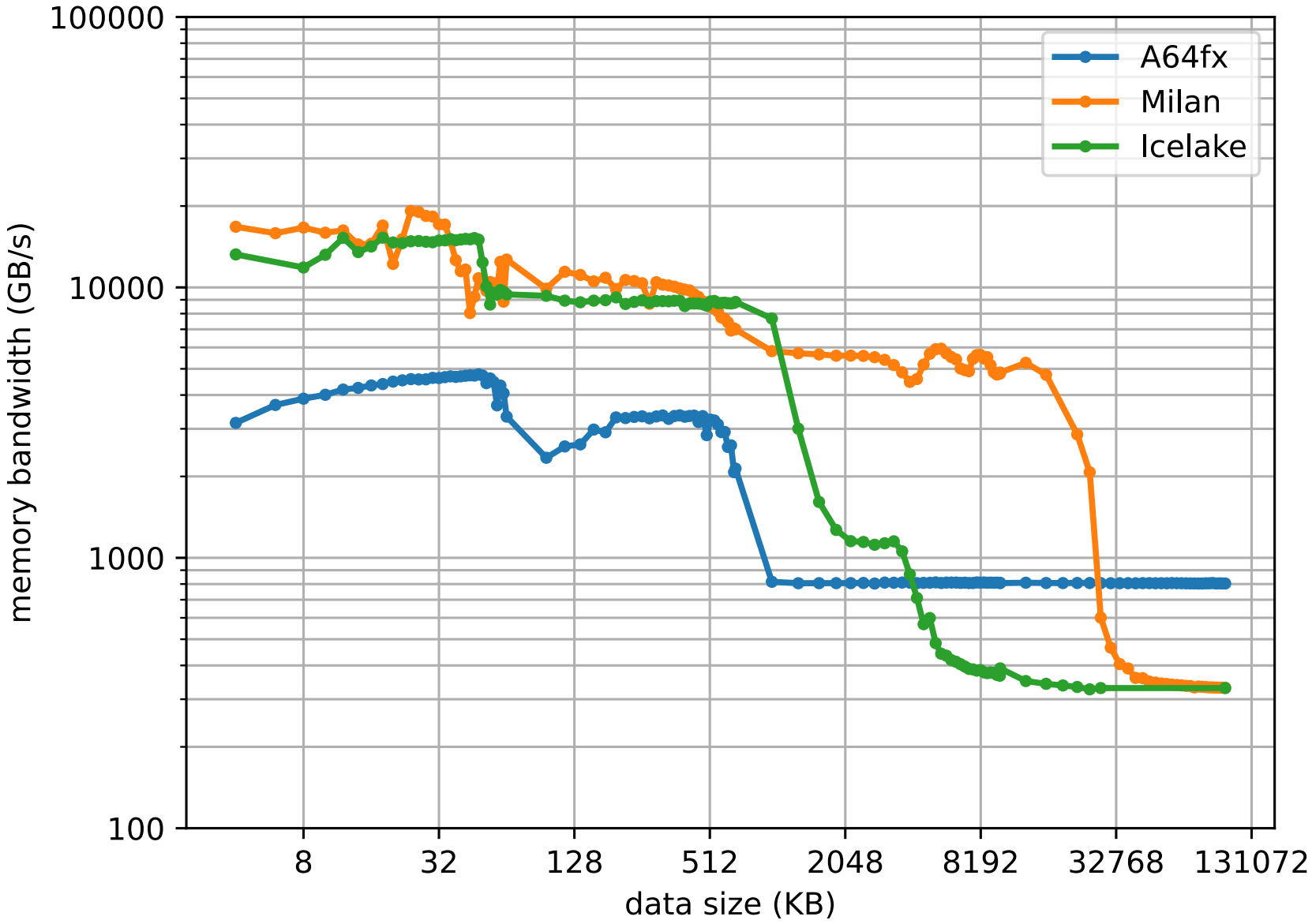
$$R = \min\left\{F_{\max}, \frac{f_d}{b_d} B_{\max}, \frac{f_d}{b_c} \beta(B_c), \right\}$$

b_c bytes moved per dof

β cache speed

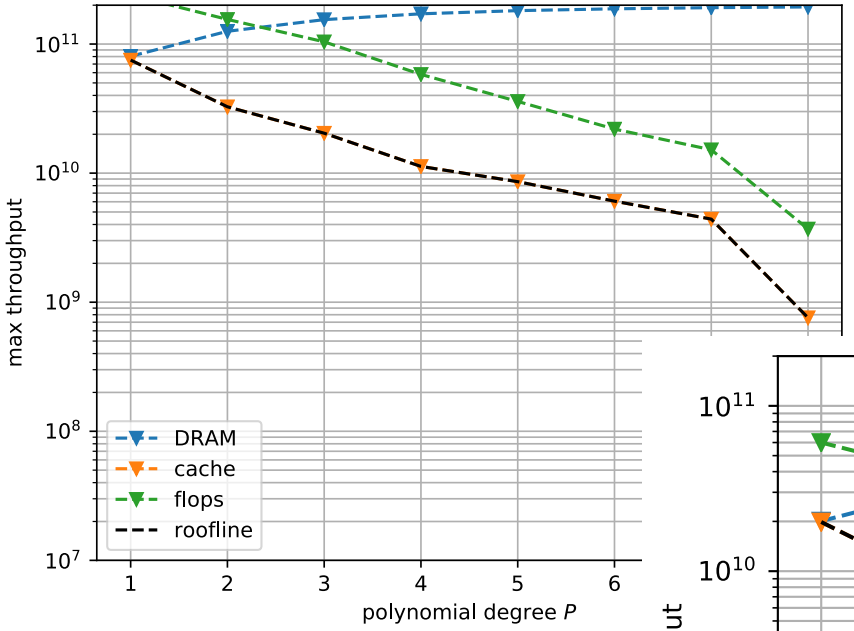
B_c size of tables and temporaries

Memory bandwidth: cache and DRAM

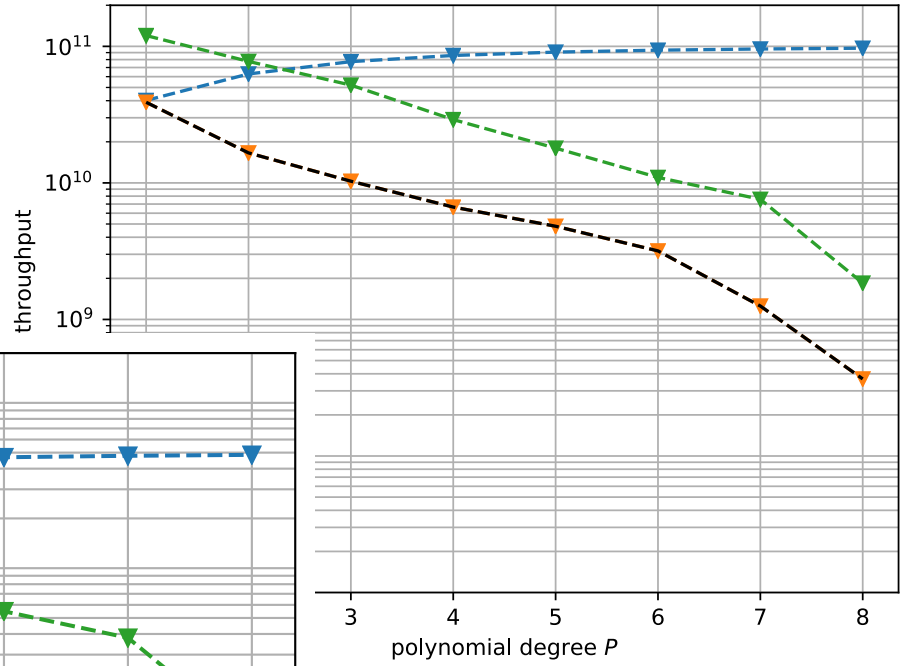


Measured using `cachebw`. More measurements ongoing.

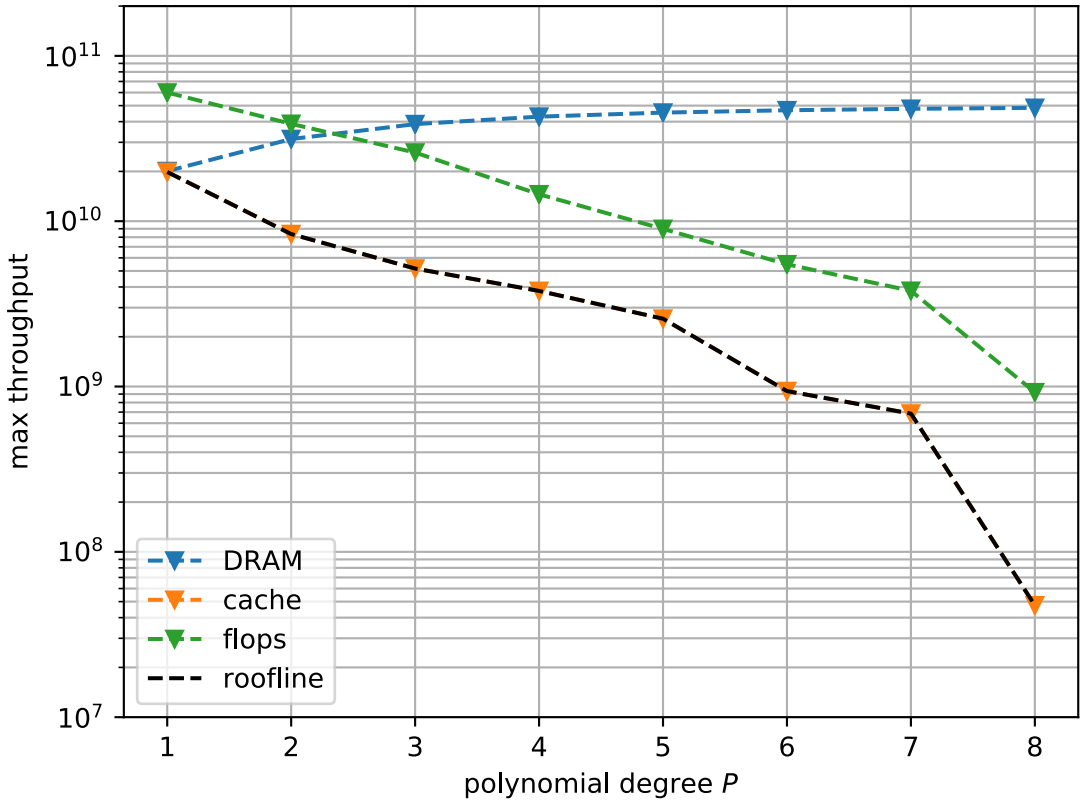
A64fx roofline: mass action, tetrahedra



half precision

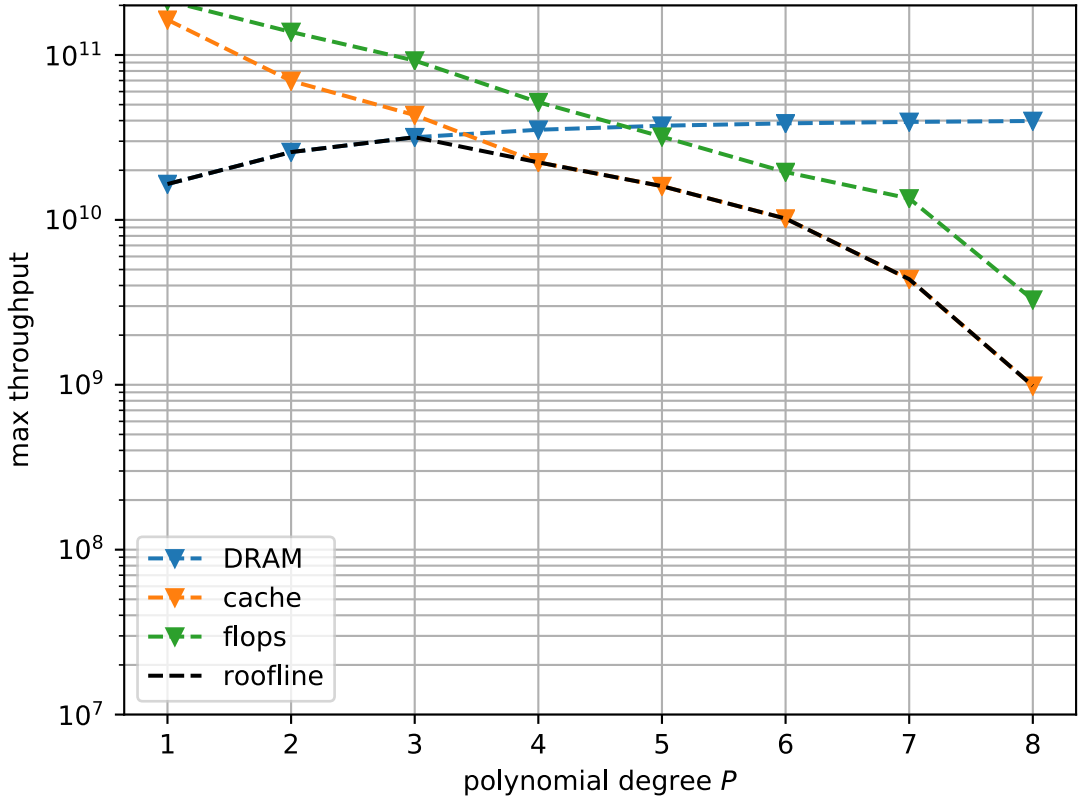


single precision

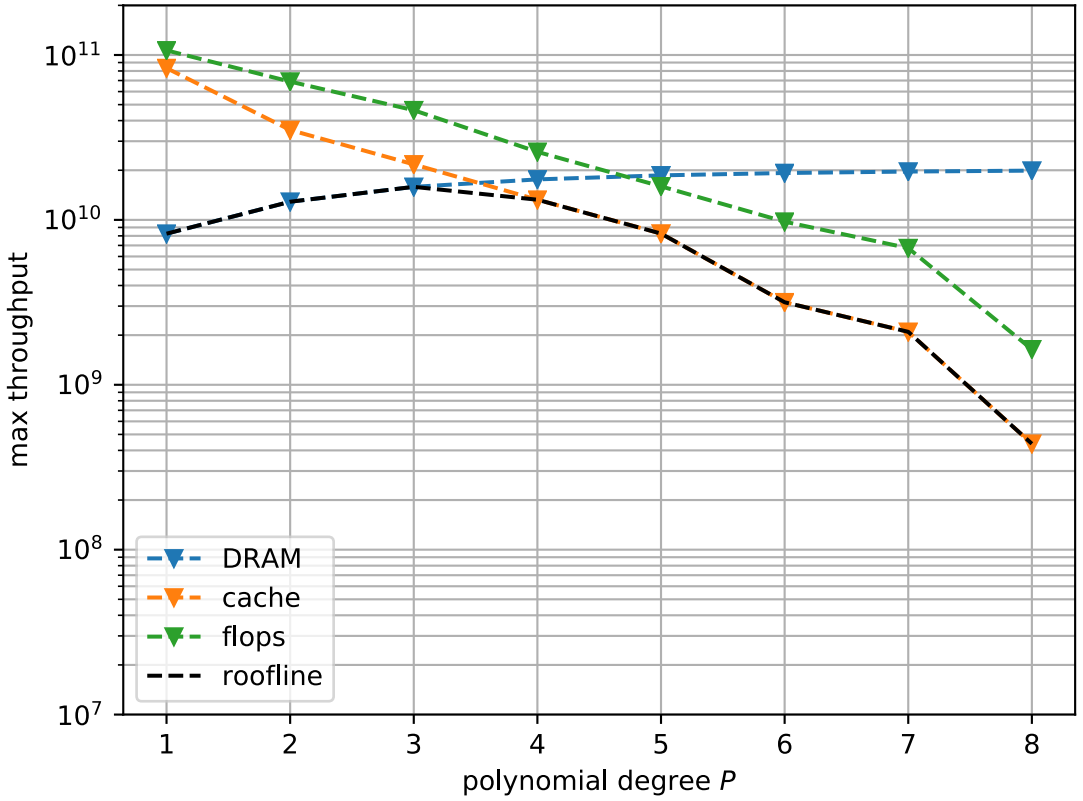


double precision

Icelake roofline: mass action, tetrahedra

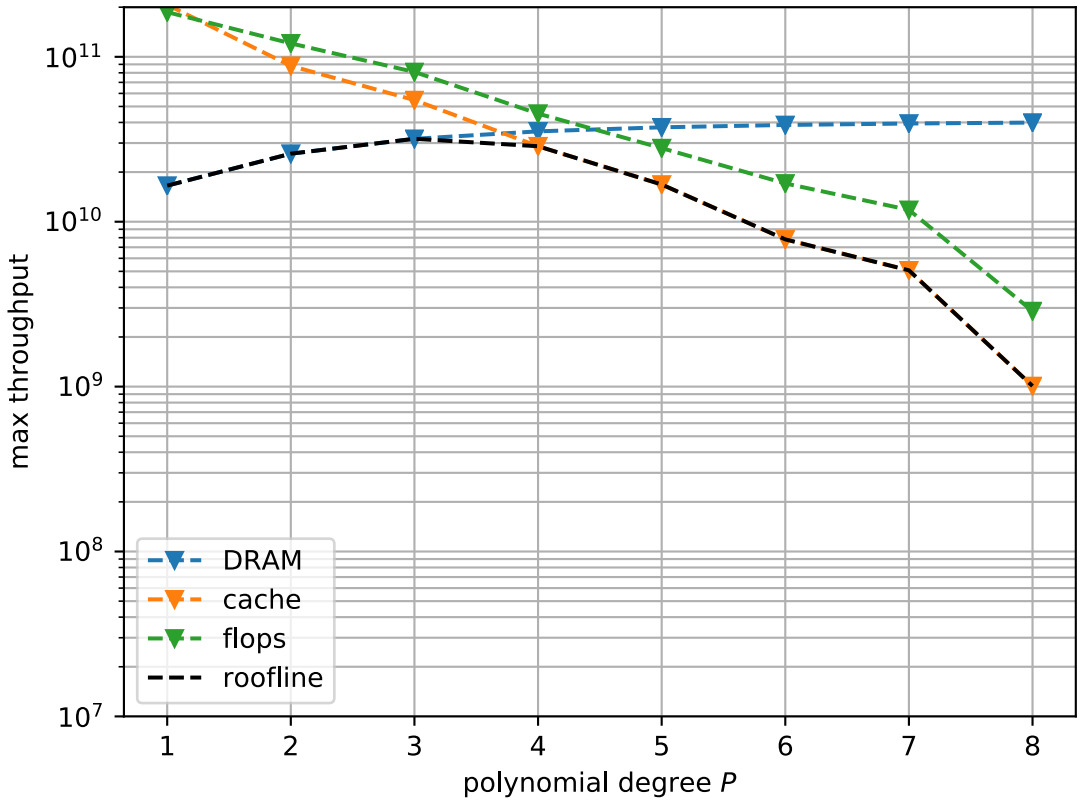


Single precision

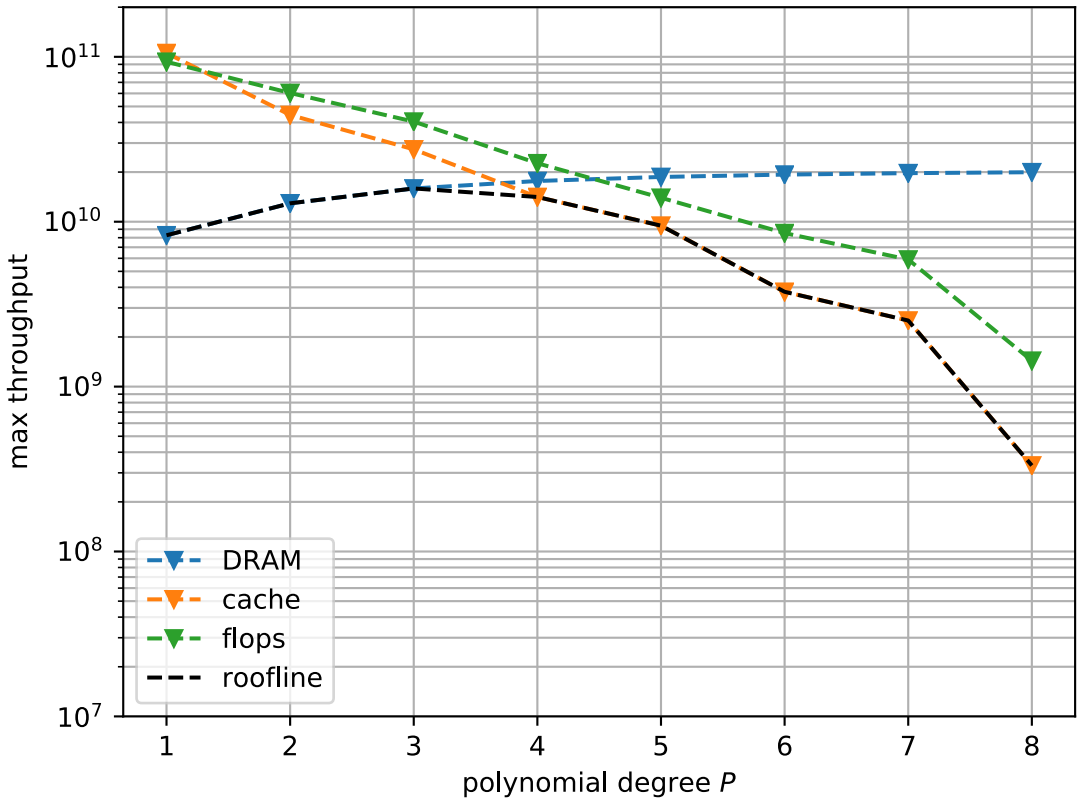


double precision

Milan roofline: mass action, tetrahedra

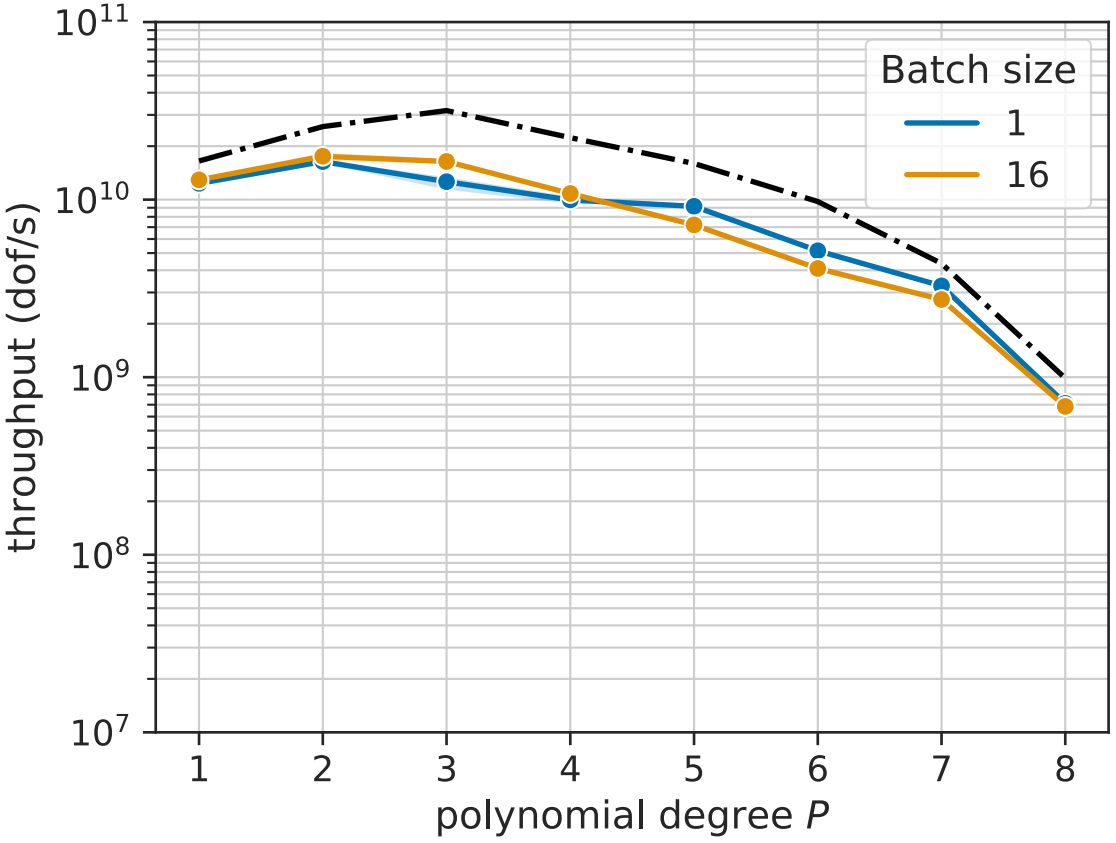


Single precision

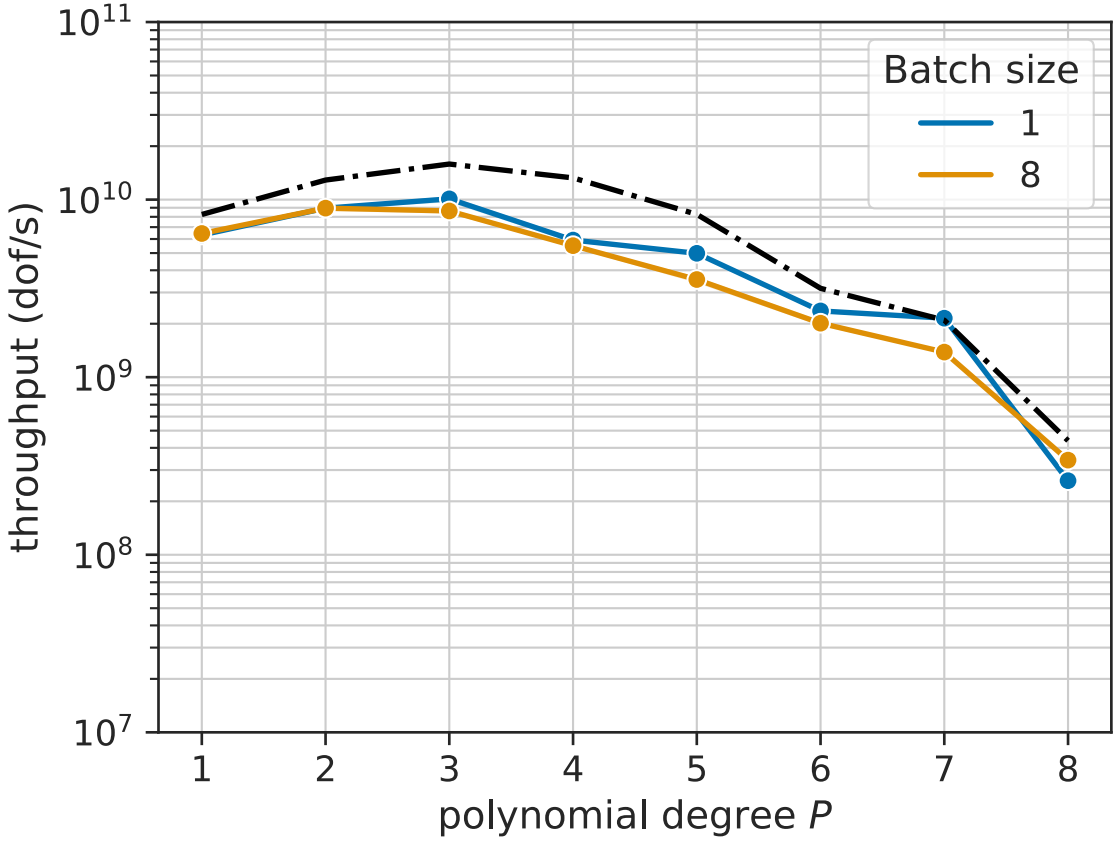


double precision

Icelake: measured performance vs model

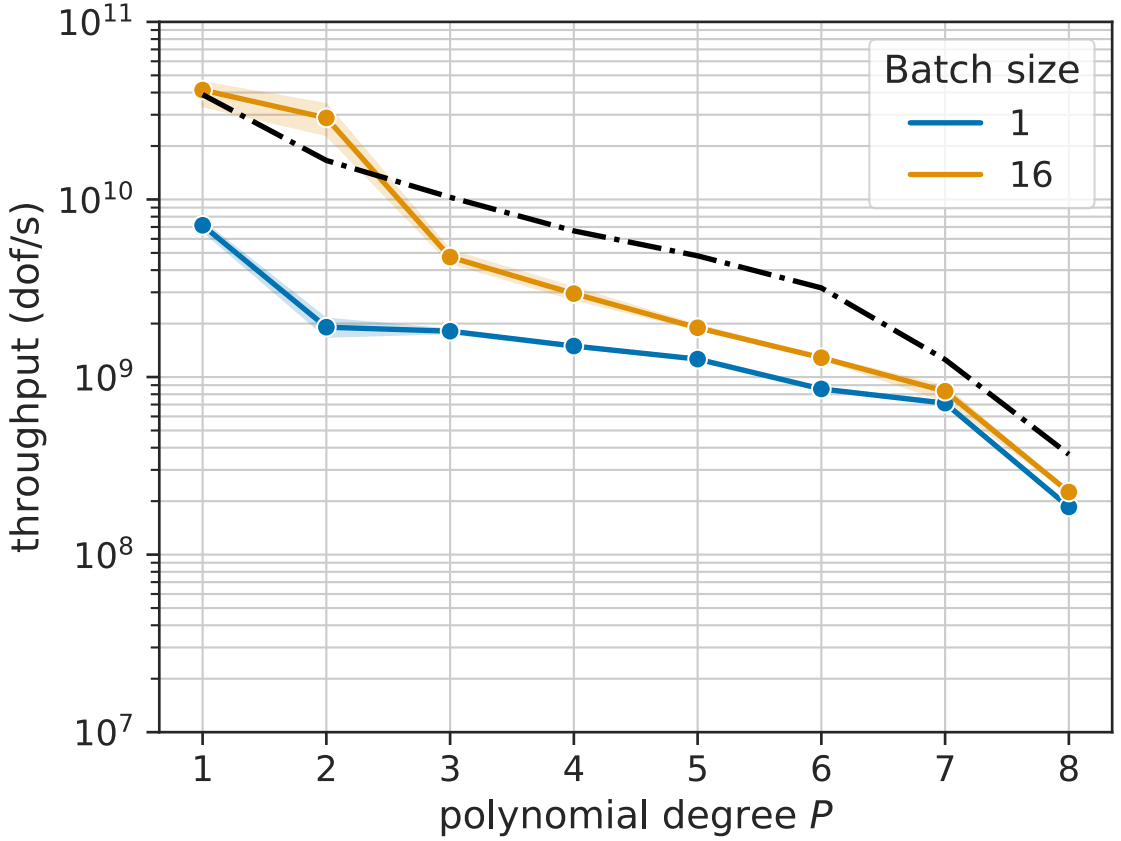


single precision

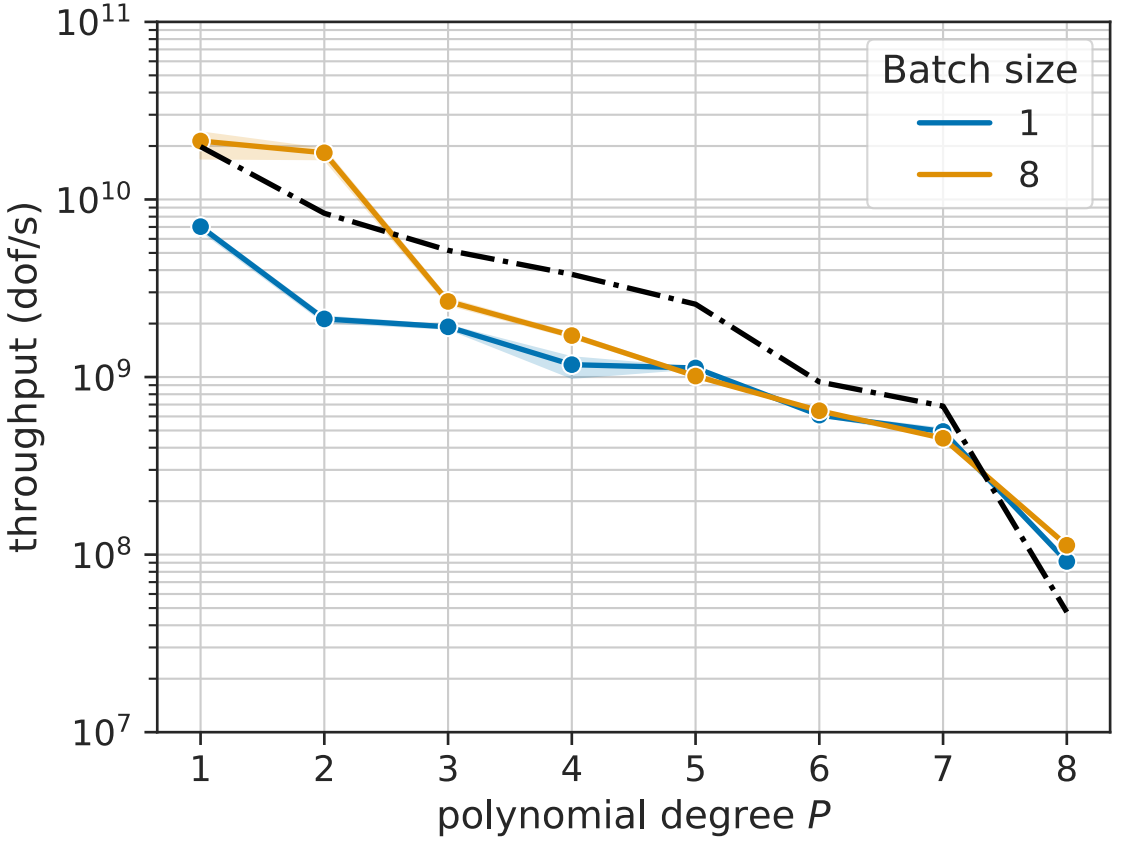


double precision

A64fx: measured performance vs model

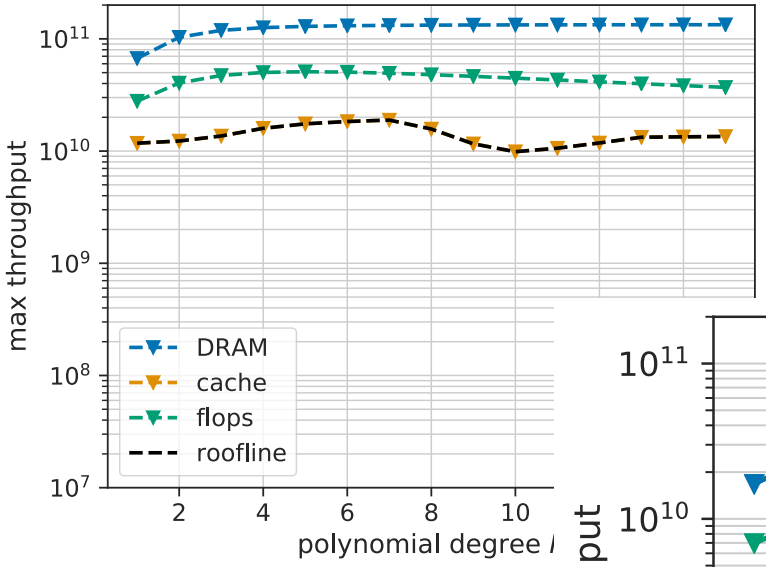


single precision

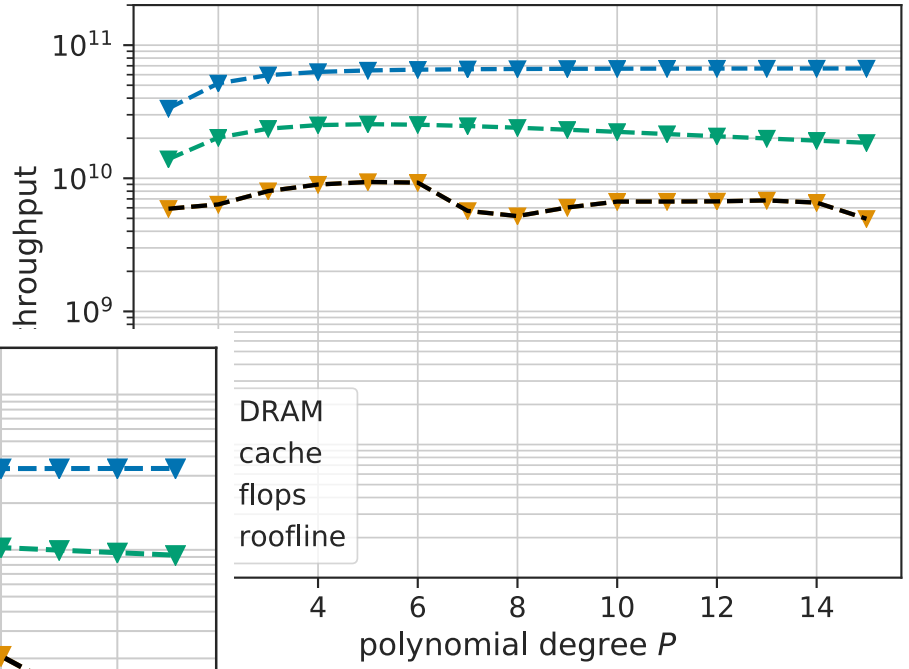


double precision

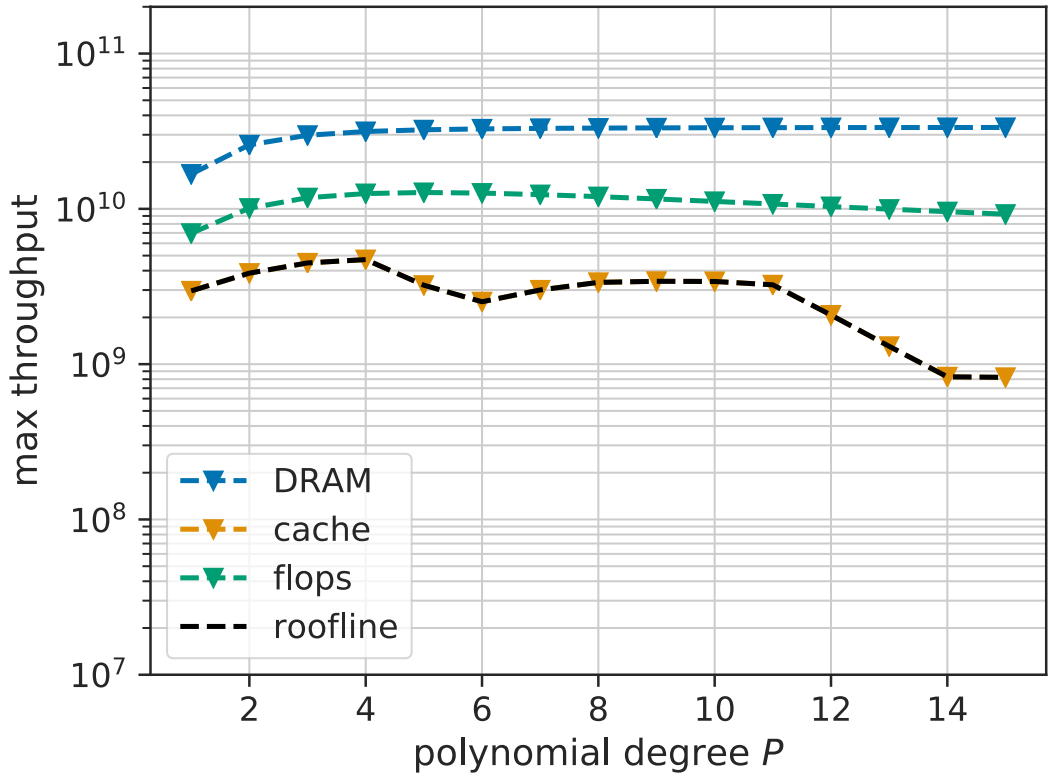
A64fx roofline: mass action, hexahedra



half precision

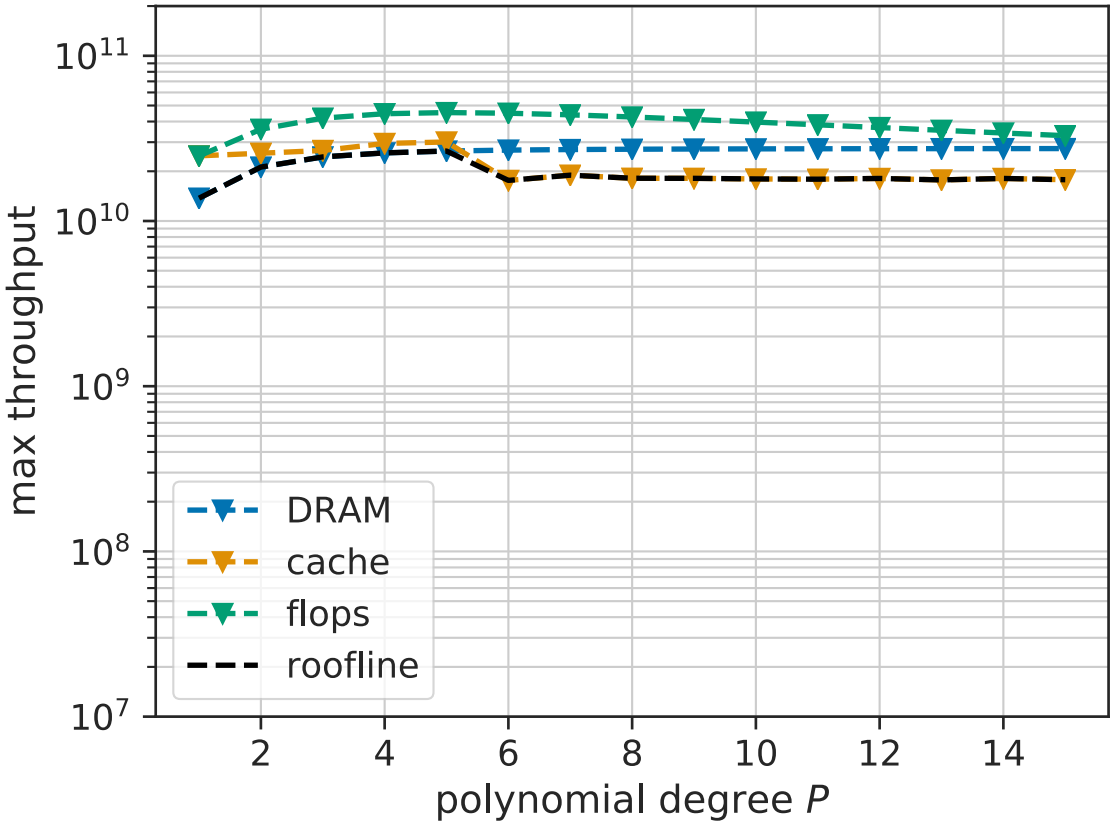


single precision

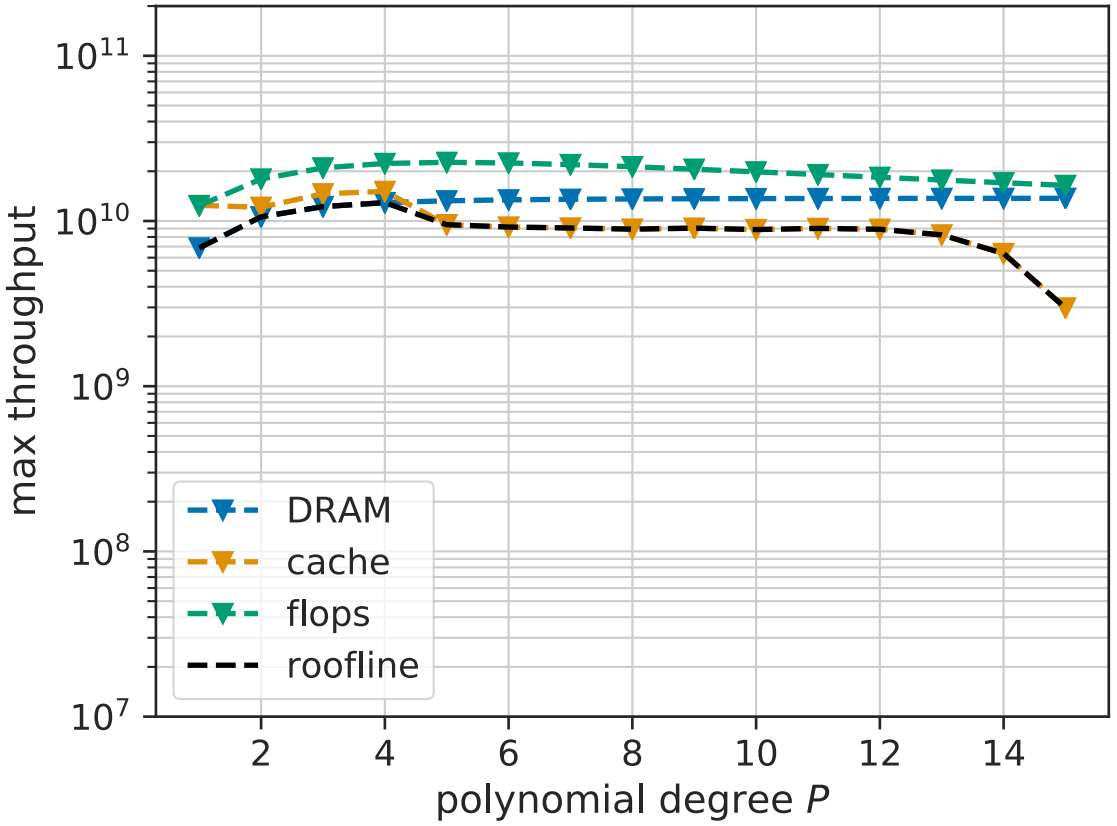


double precision

Icelake roofline: mass action, hexahedra

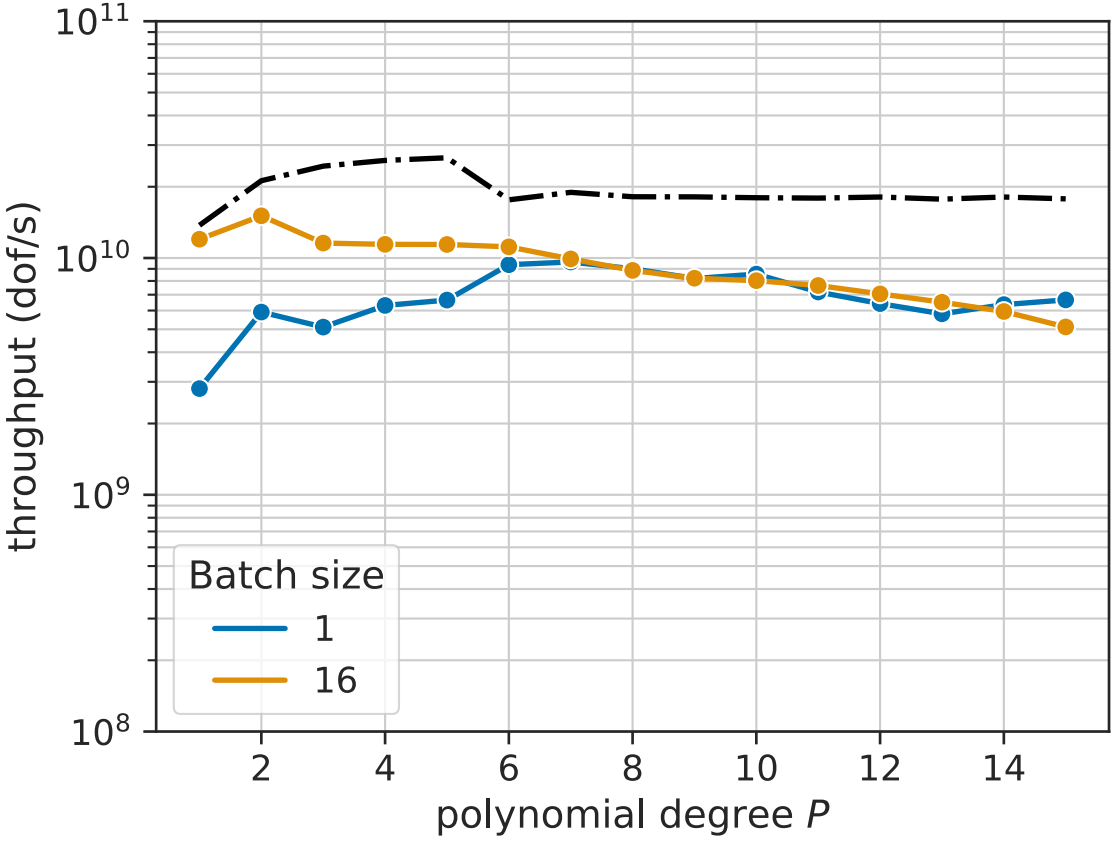


Single precision

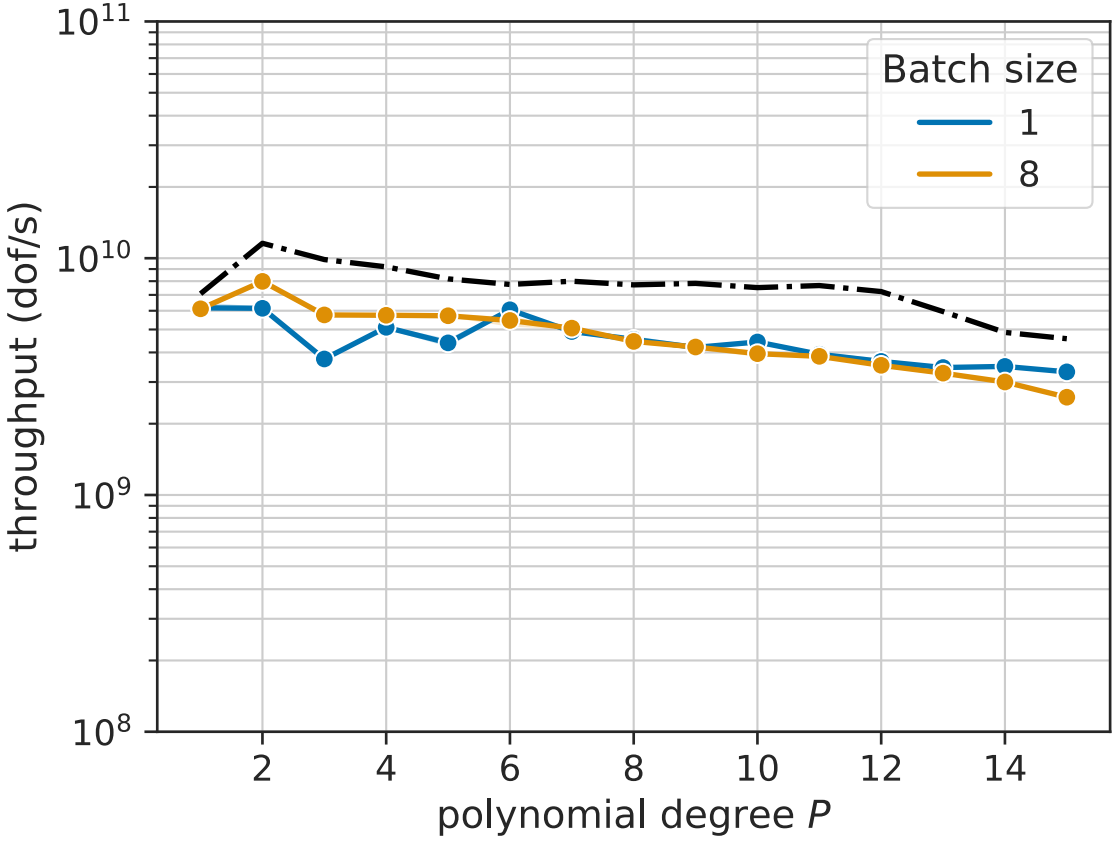


double precision

Icelake measured performance (mass action)

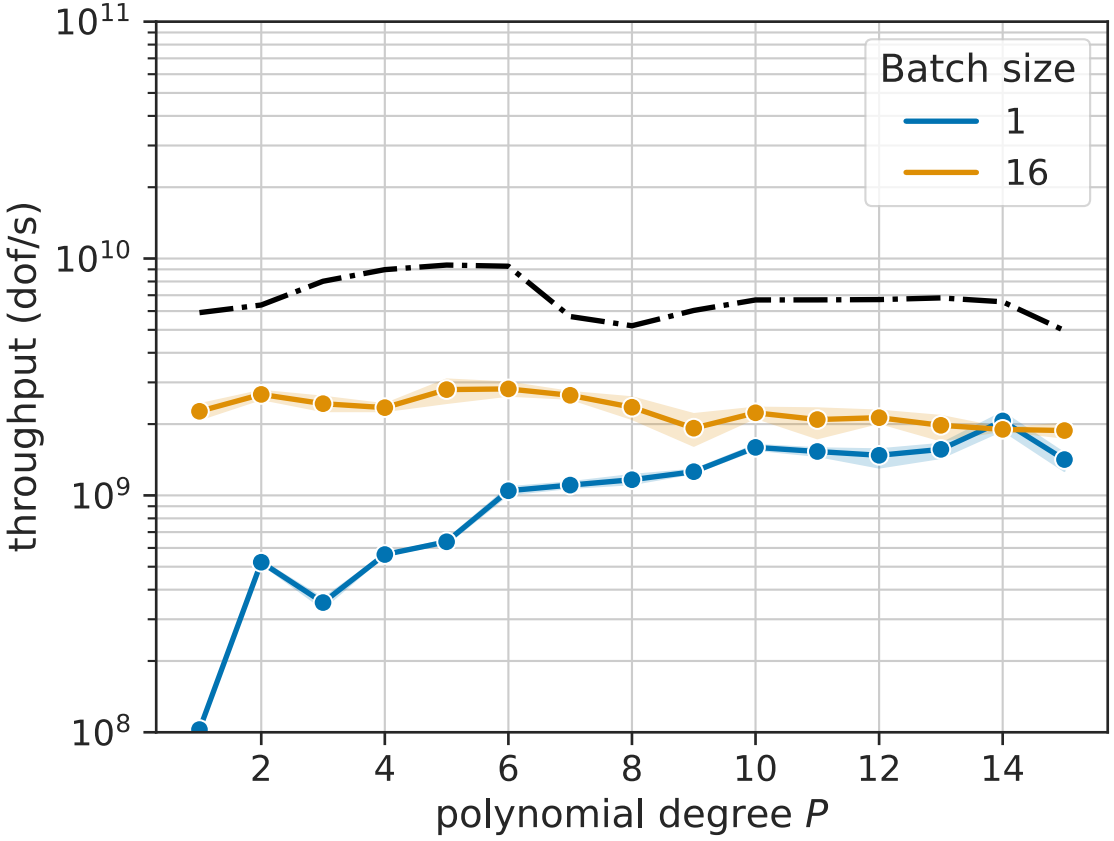


Single precision

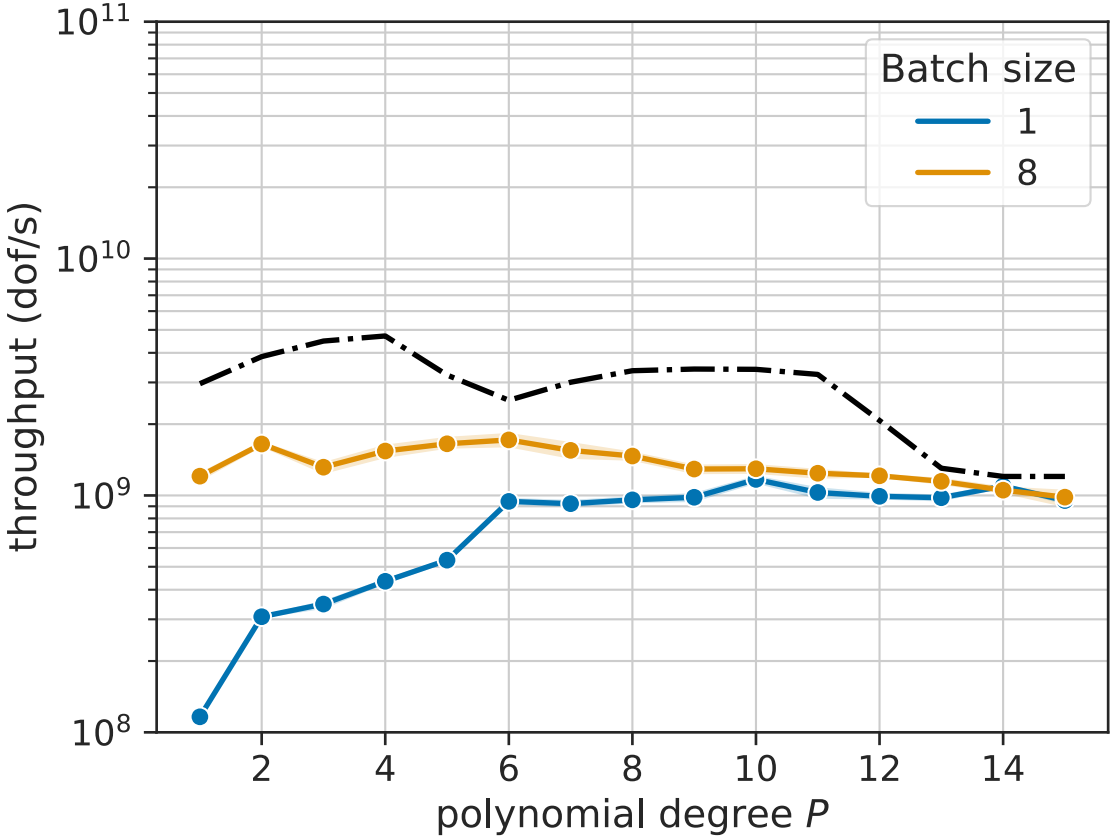


double precision

A64fx measured performance (mass action)

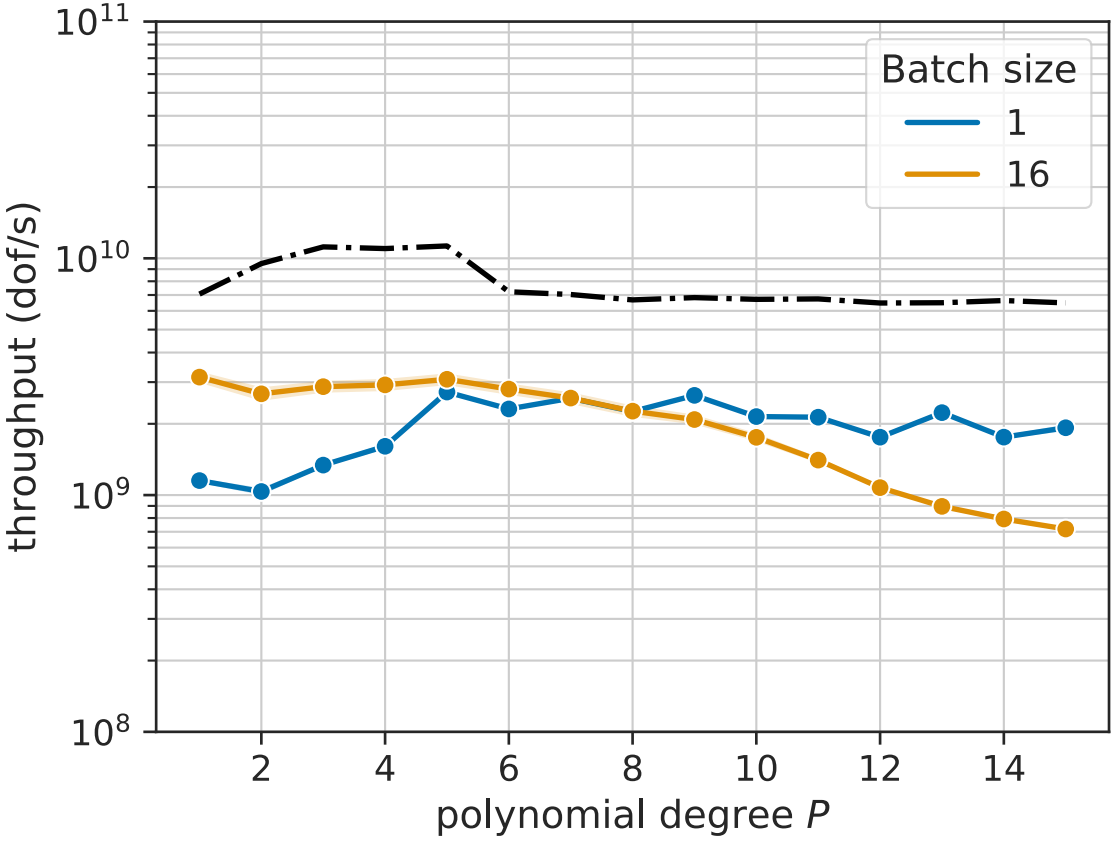


Single precision

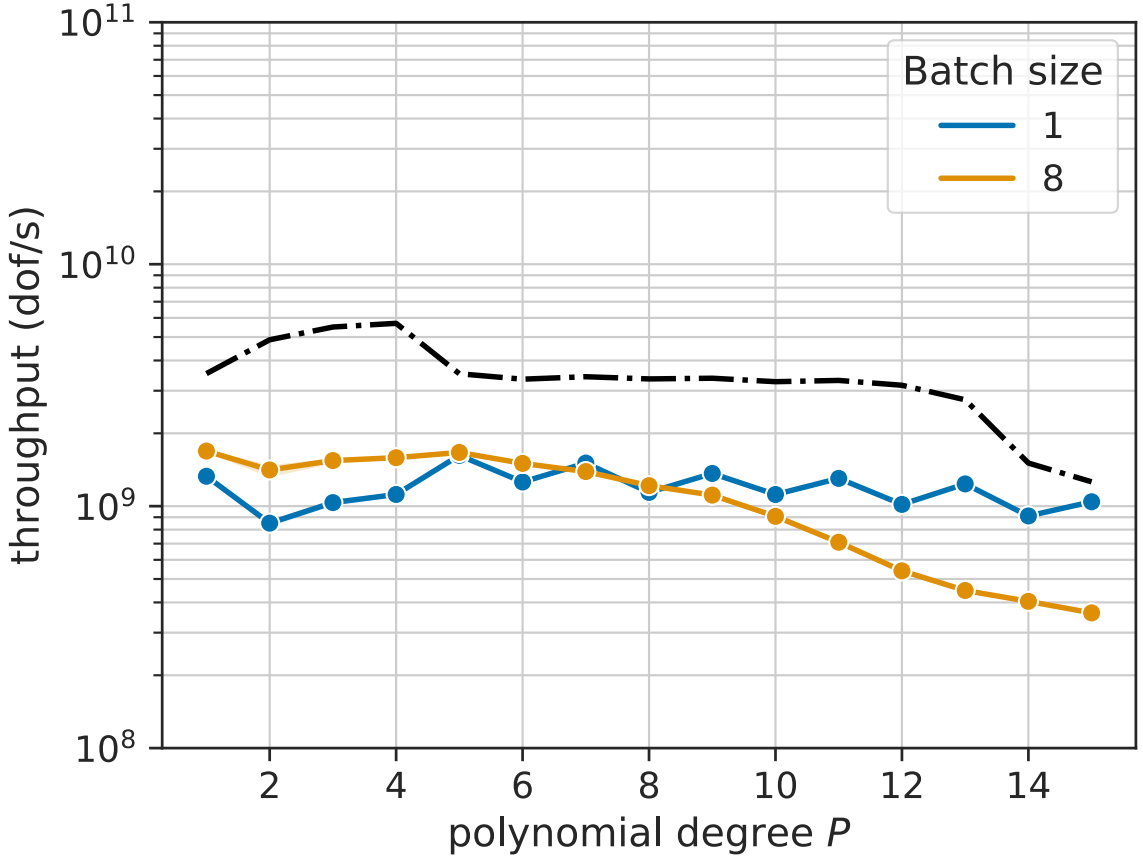


double precision

Icelake measured performance: weighted Lapalace (hex)



Single precision



double precision

Summary

- Performance of kernels typically limited by cache speed
- Cross-element vectorisation offers negligible vectorisation benefits in double precision over carefully designed single-cell kernels, but is more complex
- Cross-element vectorisation has vectorisation benefits at reduced precision for most orders
- Hexahedral cells
  - Cross-element vectorisation uses more temporaries, which can cause early cache spilling
- Simplices
  - Large tables, cross-element vectorisation reduces impact of cache spilling, esp, for *H*(div) and *H*(curl) elements
- Kernels are generated by FFCx (development branch)
- GPU work ongoing