Firedrake: automating the finite element method by composing abstractions

Lawrence Mitchell
8th June 2016

\(^1\)Departments of Computing and Mathematics, Imperial College London
Firedrake development team

IC  David A. Ham, Miklós Homolya, Fabio Luporini, Gheorghe-Teodor Bercea, Paul H. J. Kelly
Bath  Andrew T. T. McRae
ECMWF  Florian Rathgeber
Firedrake development team

IC  David A. Ham, Miklós Homolya, Fabio Luporini, Gheorghe-Teodor Bercea, Paul H. J. Kelly
Bath  Andrew T. T. McRae
ECMWF  Florian Rathgeber

www.firedrakeproject.org
Firedrake development team

David A. Ham, Miklós Homolya, Fabio Luporini, Gheorghe-Teodor Bercea, Paul H. J. Kelly
Andrew T. T. McRae
Florian Rathgeber

www.firedrakeproject.org
Rathgeber et al. 2015 arXiv:1501.01809 [cs.MS]
The right abstraction level
How do you solve the Poisson equation?

```python
from firedrake import *
mesh = UnitSquareMesh(100, 100)
V = FunctionSpace(mesh, "RT", 2)
Q = FunctionSpace(mesh, "DG", 1)
W = V*Q
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
a = dot(u, v)*dx + div(v)*p*dx + div(u)*q*dx
L = Constant(1)*v*dx
u = Function(W)
solve(a == L, u, solver_parameters={
    "ksp_type": "gmres",
    "ksp_rtol": 1e-8,
    "pc_type": "fieldsplit",
    "fieldsplit_type": "schur",
    "fieldsplit_schur_fact_type": "full",
    "fieldsplit_schur_precondition": "selfp",
    "fieldsplit_0_ksp_type": "preonly",
    "fieldsplit_0_pc_type": "ilu",
    "fieldsplit_1_ksp_type": "preonly",
    "fieldsplit_1_pc_type": "hypre"
})
```

Find $u \in V \times Q \subset H(\text{div}) \times L^2$ s.t.

$$\langle u, v \rangle + \langle \text{div } v, p \rangle = 0 \quad \forall v \in V$$

$$\langle \text{div } u, q \rangle = -\langle 1, q \rangle \quad \forall q \in Q.$$
Developing models

- Choose equations
- Pick method/discretisation
- Decide on implementation language, target architecture
- Write code
- ...
- Optimise
- ...
- Profit?
How much code do you need to change to

- Change preconditioner (e.g. ILU to AMG)?
- Drop terms in the preconditioning operator?
- Use a completely different operator to precondition?
- Do quasi-Newton with an approximate Jacobian?
- Apply operators matrix-free?
Can we offer easy experimentation without compromising performance?
Can we offer easy experimentation without compromising performance too much?
# x the input fields (e.g. current guess)
def form_residual(x):
    x_l <- x # global to ghosted
    for each element in mesh:
        x_e <- x_l[element] # gather through element map
        for each qp in element:
            basis_fns <- eval_basis_funs(qp)
            J <- compute Geometry(element, qp)
            for each bf in basis_fns:
                x_e_qp <- eval(x_e at qp)
                f_qp <- user_evaluation(qp, bf, x_e_qp)
                # insert into element residual
                f_e <- transform_to_physical(f_qp, J)
                f_l <- f_e # scatter through element map
    f <- f_l # ghosted to global
f_qp <- user_evaluation(qp, bf, x_e_qp)

- Problem-specific variability in *innermost* loop
- Efficient implementation may need to:
  - vectorize across elements,
  - vectorize within an element,
  - exchange loop orders,
  - hoist loop-invariant code,
  - exploit structure in basis functions,
  - pre-evaluate geometry at quad points.
  - ...
Say *what*, not *how*. 
Local kernels
We use UFL (Alnæs et al. 2014) from the FEniCS project for specifying variational problems.

A form compiler translates this to low-level executable code for evaluating the integral on an element.
An example

```python
mesh = UnitTriangleMesh()
V = FunctionSpace(mesh, "CG", 2)
u = TrialFunction(V)
v = TestFunction(V)
a = u*v*dx

void integral(double A[6][6],
               const double *restrict coords[6])
{
    double t0 = (-1 * coords[0][0]);
    double t1 = (-1 * coords[3][0]);
    /* t2 ← |det| */
    double t2 = fabs(((t0 + (1 * coords[1][0])) * 
                      (t1 + (1 * coords[5][0]))) +
                      (-1 * (t0 + (1 * coords[2][0])) * 
                        (t1 + (1 * coords[4][0]))));
    static const double t3[6] = {...};
    static const double t4[6][6] = {...};
    for (int ip = 0; ip < 6; ip += 1) {
        double t5 = (t3[ip] * t2);
        for (int j = 0; j < 6; j += 1) {
            for (int k = 0; k < 6; k += 1) {
                A[j][k] += t5 * t4[ip][j] * t4[ip][k];
            }
        }
    }
}
```
Two-stage form compilation

1. Lower finite element expressions to tensor-algebra
1. Lower finite element expressions to tensor-algebra
2. Lower tensor algebra to unscheduled loop nest of scalar expressions.
Two-stage form compilation

1. Lower finite element expressions to tensor-algebra
2. Lower tensor algebra to unscheduled loop nest of scalar expressions.
3. Apply optimisation passes to minimise operation count, make code amenable to vectorising compilers.
Two-stage form compilation

1. Lower finite element expressions to tensor-algebra
2. Lower tensor algebra to unscheduled loop nest of scalar expressions.
3. Apply optimisation passes to minimise operation count, make code amenable to vectorising compilers.

github.com/firedrakeproject/tsfc
Given, $c_r$, basis coefficients on a cell
Want $c_q$, coefficient evaluated at quad points.

$$c_q = \sum_{r} \mathcal{E}_{q,r} c_r$$

$\mathcal{E}_{q,r}$ provided by FIAT as tabulation of 2D array.
Lowering FE

Given, $c_r$, basis coefficients on a cell
Want $c_q$, coefficient evaluated at quad points.

$$c_q = \sum_r \mathcal{E}_{q,r} c_r$$

$\mathcal{E}_{q,r}$ provided by FIAT as tabulation of 2D array.

**Structure in $\mathcal{E}$**

$$\mathcal{E}_{q,r} = \mathcal{E}_{(q_x,r_x),(q_y,r_y)}$$

$\mathcal{E}$ factorises

$$\mathcal{E}_{q,r} = \mathcal{E}^x_{q_x,r_x} \mathcal{E}^y_{q_y,r_y}$$

WIP: exploiting structure for automated sum-factorisation.
Problem

Modern optimising compilers do a bad job on finite element kernels.
Optimisation of finite element kernels

Problem

*Modern optimising compilers do a bad job on finite element kernels.*

Code motion (or not?)

```c
for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
        for (k = 0; k < N; k++)
            A[j][k] += f(i, j)*g(i, k)
```
Problem

*Modern optimising compilers do a bad job on finite element kernels.*

**Code motion (or not?)**

```c
for (i = 0; i < L; i++)
    for (j = 0; j < M; j++)
        for (k = 0; k < N; k++)
            A[j][k] += f(i, j)*g(i, k)
```

**Corollary**

*We need to spoon-feed the compiler already optimised code.*
No single optimal schedule for evaluation of every finite element kernel. Variability in

- polynomial degree,
- number of fields,
- kernel complexity,
- working set size,
- structure in the basis functions,
- structure in the quadrature points,
- ...

We explore (some of) this space using a special-purpose compiler.
Vectorisation
Align and pad data structures, then use intrinsics or rely on compiler. Experience, gcc-4.X really doesn’t want to vectorise short loops!

Flop reduction
Exploit linearity in test functions to perform factorisation, code motion and CSE.
Cost model: don’t introduce mesh-sized temporaries.

github.com/coneoproject/COFFEE
Global iteration
A library for expressing data parallel iterations

**Sets**  iterable entities

**Dats**  abstract managed arrays (data defined on a set)

**Maps**  relationships between elements of sets

**Kernels**  local computation

**par_loop**  Data parallel iteration over a set

Arguments to parallel loop indicate how to gather/scatter global data using *access descriptors*

```
par_loop(kernel, iterset, data1(map1, READ), data2(map2, WRITE))
```
Key ideas

Local computation
Kernels do not know about global data layout.

- Kernel defines contract on local, packed, ordering.
- Global-to-local reordering/packing appears in map.

“Implicit” iteration
Application code does not specify explicit iteration order.

- Define data structures, then just “iterate”
- Can’t write Gauss-Seidel (for example)
- Lazy evaluation
Performance

- Keep data in cache as long as possible.
- Manually fuse kernels.
- Loop tiling for latency hiding.
- ...
- Individual components hard to test
- Space of optimisations suffers from combinatorial explosion.
Maintainability

- Keep kernels separate
- “Straight-line” code
- ...
- Testable
- Even if performance of individual kernels is good, can lose *a lot*
• `par_loop` only executed “when you look at the data”.
• PyOP2 sees sequence of loops, can reason about them for
  • Loop fusion
  • Loop tiling
    • Communication coalescing
• Application code does not change. “What, not how”.
Another example: extruded meshes

In many geophysical applications, meshes are topologically *structured* in the (short) vertical direction.

Can produce vertically structured dof-numbering, avoiding most of the indirection penalty.

Only need to annotate data structures with this extra information.

```python
s = Set(100)  # Unstructured set
e = ExtrudedSet(s, layers=...)  # Semi-structured set
d = DataSet(...)  
# values encode indirections on base set
# offsets say how to move in the structured direction
map = Map(e, d, values, offsets=[...])

par_loop(kernel, e, data(map, WRITE), ...)
```
<table>
<thead>
<tr>
<th>Number of layers</th>
<th>Performance [GFLOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1 CORE (2.6 GFLOPS)</td>
</tr>
<tr>
<td>20.0</td>
<td>BALANCED (5.2 GFLOPS)</td>
</tr>
<tr>
<td>40.0</td>
<td>16 CORES (41.6 GFLOPS)</td>
</tr>
<tr>
<td>60.0</td>
<td>E5-2640 Xeon Haswell v3</td>
</tr>
<tr>
<td>80.0</td>
<td>CG1xCG1</td>
</tr>
<tr>
<td></td>
<td>CG1xDG0</td>
</tr>
<tr>
<td></td>
<td>CG1xDG1</td>
</tr>
<tr>
<td></td>
<td>DG0xCG1</td>
</tr>
<tr>
<td></td>
<td>DG0xDG0</td>
</tr>
<tr>
<td></td>
<td>DG0xDG1</td>
</tr>
<tr>
<td></td>
<td>DG1xCG1</td>
</tr>
<tr>
<td></td>
<td>DG1xDG0</td>
</tr>
<tr>
<td></td>
<td>DG1xDG1</td>
</tr>
</tbody>
</table>

Did we succeed?
With model set up, experimentation is easy

- Change preconditioner: c. 1 line
- Drop terms: c. 1-4 lines
- Different operator: c. 1-10 lines
- quasi-Newton: c. 1-10 lines
- Matrix-free: XXX
## Maintainability

### Core Firedrake

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firedrake</td>
<td>9000</td>
</tr>
<tr>
<td>PyOP2</td>
<td>5000</td>
</tr>
<tr>
<td>TSFC</td>
<td>2700</td>
</tr>
<tr>
<td>COFFEE</td>
<td>4500</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>21200</strong></td>
</tr>
</tbody>
</table>

### Shared with FEniCS

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIAT</td>
<td>4000</td>
</tr>
<tr>
<td>UFL</td>
<td>13000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>17000</strong></td>
</tr>
</tbody>
</table>
Kernel performance

- COFFEE produces kernels that are better (operation count) than existing automated form compilers
- Provably optimal in some cases
- Also provides good vectorised performance, up to 70% peak for in-cache computation.
Thetis

- 3D unstructured coastal ocean model in Firedrake
- Lock exchange test case

**Thetis** P1DG-P1DG, triangular wedges. 13s/s.

**SLIM** hand-coded/optimised (same discretisation), 6s/s

github.com/thetisproject/thetis
Exposing PyOP2 provides means of writing mesh iterations that are not “assemble a variational form”.

**Slope limiters**

Vertex-based limiters need max/min over incident cells

```c
par_loop("""
    for (int i=0; i<qmax.dofs; i++) {
        qmax[i][0] = fmax(qmax[i][0], centroids[0][0]);
        qmin[i][0] = fmin(qmin[i][0], centroids[0][0]);
    }
"""",
    dx,
    { 'qmax': (max_field, RW),
      'qmin': (min_field, RW),
      'centroids': (centroids, READ) })
```
• Efficient high order evaluation (via tensor-products and/or Bernstein polynomials)
• Matrix-free operators and preconditioning
• Symbolic (not just algebraic) composition for multiphysics preconditioning
• Mesh adaptivity
• ...

Coming (soon)
Summary

- Firedrake provides a layered set of abstractions for finite element
- Computational performance is good, often > 50% achievable peak.
- Hero-coding necessary if you want the last 10-20%
- ...but at what (person) cost.
Questions?


