Firedrake: automating the finite element method by composing abstractions

Lawrence Mitchell
26th January 2017

1Departments of Computing and Mathematics, Imperial College London
Firedrake team

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

www.firedrakeproject.org
Contact

www.firedrakeproject.org/contact.html

Methods

- Slack: firedrakeproject.slack.com
- Mail: firedrake@imperial.ac.uk (subscribe first)
- Github: github.com/firedrakeproject/firedrake
What is Firedrake?

[...] an automated system for the portable solution of partial differential equations using the finite element method.

- Written in Python.
- Finite element problems specified with *embedded* domain specific language.
- *Runtime* compilation to low-level (C) code.
- Expressly *data parallel*: don’t worry about MPI.
A specification of finite element problems

```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",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "full",
    "pc_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.

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

$$\langle \text{div} u, q \rangle = -\langle 1, q \rangle \quad \forall q \in Q.$$
Symbolic, numerical computing

Weave together

- **symbolic** problem description
  
  \[
  W = V*Q \\
  u, p = \text{TrialFunctions}(W) \\
  v, q = \text{TestFunctions}(W) \\
  a = \text{dot}(u, v)*dx + \text{div}(v)*p*dx + \text{div}(u)*q*dx \\
  L = -\text{Constant}(1)*v*dx
  \]

- with problem-specific data (which mesh, what solver?)
  
  \[
  \text{mesh} = \text{UnitSquareMesh}(100, 100) \\
  V = \text{FunctionSpace}(\text{mesh}, "RT", 2) \\
  Q = \text{FunctionSpace}(\text{mesh}, "DG", 1) \\
  \ldots \\
  \text{solve}(a == L, u, \text{solver\_parameters}=\ldots)
  \]

and *synthesize* efficient implementation from the symbolic problem description.
More than a pretty face

Library usability

• High-level language enables rapid model development
• Ease of experimentation
• Small model code base

Library development

• Automation of complex optimisations
• Exploit expertise across disciplines
• Small library code base
Composability of libraries that manipulate PDE solvers

www.dolfin-adjoint.org
Automated derivation of the discrete adjoint from forward models written using FEniCS and Firedrake.

$cloc dolfin-adjoint/
Language  files  blank  comment  code
Python    54     2322  937  7294
$cloc dolfin-adjoint/compatibility.py
Python    1      38   11  140
Ease of experimentation

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?

Same “easy to use” code must run fast at scale.
Say *what*, not *how*.
Local kernels
Automating expertise

- “In-person” case-by-case optimisation does not scale
- Code generation allows us to package expertise and provide it to everyone
- Done by a special-purpose kernel compiler
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,
- ...
Vectorisation
Align and pad data structures, then use intrinsics or rely on compiler.

Flop reduction
Exploit linearity in test functions to perform factorisation, code motion and CSE.
github.com/coneoproject/COFFEE
Global iteration
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
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”
- Lazy evaluation
Did we succeed?
Experimentation

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: c. 1-10 lines (+ c. 30 lines for preconditioner).
### Core Firedrake

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firedrake</td>
<td>11000</td>
</tr>
<tr>
<td>PyOP2</td>
<td>5000</td>
</tr>
<tr>
<td>TSFC</td>
<td>3500</td>
</tr>
<tr>
<td>COFFEE</td>
<td>4500</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>24000</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
- Good vectorised performance, problem dependent, but up to 70% peak for in-cache computation.
Summary

- Firedrake provides a layered set of abstractions for finite element
- Enables automated provision of expertise to model developers
- Computational performance is good, often > 50% achievable peak.
