Firedrake: a multilevel domain specific language approach to unstructured mesh stencil computations

Lawrence Mitchell, Gheorghe-Teodor Bercea, David Ham, Paul Kelly, Nicolas Lornant, Fabio Luporini, Florian Rathgeber

Departments of Mathematics and Computing, Imperial College London

21 February 2014
Introduction

Maintaining abstractions

Exploiting structure

Benchmarking

Conclusions
What are we interested in?

- (Predominantly) finite element simulations
  - primary application areas in geophysical fluids (ocean and atmosphere)
  - simulations on unstructured and semi-structured meshes
- Providing high-level interfaces for users, with performance
What are we interested in?

- (Predominantly) finite element simulations
  - primary application areas in geophysical fluids (ocean and atmosphere)
  - simulations on unstructured and semi-structured meshes
- Providing high-level interfaces for users, with performance
- the moon, on a stick
What are the elementary operations?

- **Numerics** tell us the elementary operation we apply everywhere in the mesh (a "kernel")
- **Mesh topology** gives us the "stencil" pattern
- Our job: efficiently apply the kernel over the whole mesh
Introduction

Maintaining abstractions

Exploiting structure

Benchmarking

Conclusions
Express what, not how

- User code should make as few decisions about implementation as possible
- FE discretisations expressed symbolically using the **Unified Form Language**
  - developed in the FEniCS project ([http://www.fenicsproject.org](http://www.fenicsproject.org))
  - symbolic representation compiled to a C kernel
- Data to feed to kernel (and interface to solvers) provided by Firedrake ([http://www.firedrakeproject.org](http://www.firedrakeproject.org))
- Execution of kernel over entire domain expressed as parallel loop with access descriptors
  - uses PyOP2 unstructured mesh library ([http://github.com/OP2/PyOP2](http://github.com/OP2/PyOP2))
PyOP2
Parallel unstructured mesh computation framework

Firedrake
A performance-portable Finite-element computation framework

Geometry, fields and meshes

Problem definition in FEM weak form

FFC Form Compiler
Local assembly kernels, data dependencies

Unified Form Language (UFL)

PyOP2 Interface

PyOP2
Parallel unstructured mesh computation framework

Parallel loops over kernels with access descriptors

Parallel scheduling, code generation

CPU (+MPI + OpenMP/OpenCL)

Explicitly parallel hardware-specific implementation

GPU (CUDA/OpenCL)

Future arch.
An example

```python
from firedrake import *
m = UnitSquareMesh(32, 32)
V = FunctionSpace(m, 'Lagrange', 2)
u = Function(V)
v = TestFunction(V)
# F(u; v) = \int \nabla u \cdot \nabla v + uvdx
F = inner(grad(u), grad(v))*dx + u*v*dx
solve(F == 0, u)
```

- Kernels produced for residual and jacobian evaluation
  - Jacobian computed by symbolic differentiation of residual form
- Kernels executed over mesh using PyOP2
  - [http://github.com/OP2/PyOP2](http://github.com/OP2/PyOP2)
PyOP2 data model

- Data types
  - **Set** e.g. cells, degrees of freedom (dofs)
  - **Dat** data defined on a Set (one entry per set element)
  - **Map** a mapping between two sets (e.g. cells to dofs), a "stencil"
  - **Global** global data (one entry)
  - **Kernel** a piece of code to execute over the mesh (in C)

- access descriptors
  - READ, RW, WRITE, INC, ...

- iteration construct
  - **par_loop** execute a Kernel over every element in a Set
Example

```python
elements = Set(...)  nodes = Set(...)  
elem_node = Map(elements, nodes, 3, ...)  # 3 nodes per element
node_data = Dat(nodes, ...)  
element_data = Dat(elements, ...)  
count = Global(...)  # no set (global value)
par_loop(kernel, elements,
    element_data(READ),     # direct read
    node_data(INC, elem_node),  # indirect increment
    count(INC))  # global increment
```

- executes `kernel` for each `ele` in `elements`
- runtime knows it has to care about data dependencies for
  - increments into `node_data`
  - increment into `count`
Synthesis, not analysis

- Low level code is generated at runtime for parallel loops
- Access descriptors on parallel loops mean:
  - code generation requires synthesis, not analysis
  - determination of when halo exchanges need to occur is automatic
  - colouring for shared memory parallelisation can be computed automatically
Synthesis, not analysis

- Low level code is generated at runtime for parallel loops
- Access descriptors on parallel loops mean:
  - code generation requires synthesis, not analysis
  - determination of when halo exchanges need to occur is automatic
  - colouring for shared memory parallelisation can be computed automatically
Introduction

Maintaining abstractions

Exploiting structure

Benchmarking

Conclusions
Many application areas have a "short" direction
  - ocean and atmosphere
  - thin shells
Numerics dictate we should do something different in short direction
Use semi-structured meshes
  - unstructured in "long" directions, structured in short
  - can we exploit this structure?
A picture of triangles
Admits a fast implementation

- Exploit structure in mesh to amortize indirect lookups
  - arrange for iteration over short direction to be innermost loop
  - pay one indirect lookup per mesh column
  - walk up column directly
Introduction

Maintaining abstractions

Exploiting structure

Benchmarking

Conclusions
A bandwidth bound test

- Walk over mesh, read from vertices and cells, sum into global

```c
void kernel(double *a, double *x[], double *y[]) {
    const double area = fabs(x[0][0]*(x[2][1]-x[4][1])
                        + x[2][0]*(x[4][1]-x[0][1])
                        + x[4][0]*(x[0][1]-x[2][1]));
    *a += area * 0.5 * y[0][0];
}
```

- Can we sustain an appreciable fraction of memory bandwidth?
Measuring throughput

- "Effective" data volume
  - assume every piece of data is touched exactly once (in perfect order)
  - don’t count data movement for indirection maps

- "Valuable" bandwidth
  - effective data volume per second

- Actual memory bandwidth will be higher (reading indirection maps)
  - but this is not "useful"
Benchmark setup

- 2D unstructured mesh: 806110 cells, 403811 vertices.
  - 2D coordinate field located at vertices (implicit 3rd coordinate)
  - Scalar field stored at cell centres
- Run with increasing number of extruded cell layers ($n_{\text{layer}}$)
  - Data volume $(806110 \times n_{\text{layer}}) + 403811 \times 2 (n_{\text{layer}} + 1)$ doubles
  - 1 layer: 18.4MB
  - 200 layers: 2468MB
- Execute kernel over mesh 100 times
Single node

- Intel Sandybridge 4 cores (2 way hyperthreading)
  - 32kB L1 cache (per core)
  - 256 kB L2 cache (per core)
  - 8 MB L3 cache (shared)
- Measured STREAM bandwidth (8 threads)
  - 11341 MB/s
Effect of good base numbering

- Being completely unstructured hurts a lot
- Compare default (mesh generator) numbering with renumbered mesh using 2D space filling curve

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Valuable bandwidth/(MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1306</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>4000</td>
</tr>
</tbody>
</table>

- Original numbering (1 layer)
- Good numbering (1 layer)

Imperial College London
Adding layers amortizes indirection cost

- L3 cache bandwidth
  - low layer numbers hit the L3 more often (indirection lookups)

- What about actual throughput though?
Above ~20 layers, indirection cost "hidden"
More threads

- Hyperthreading gives some further gains (82% STREAM bandwidth)
Introduction

Maintaining abstractions

Exploiting structure

Benchmarking

Conclusions
Possible to be unstructured and fast

▶ A good numbering gets you a reasonable way there
▶ If there is structure in your problem, use it!
▶ High level abstractions need not kill performance
All code is open source, and online:

Firedrake  http://www.firedrakeproject.org and
http://github.com/firedrakeproject/firedrake

PyOP2  http://github.com/OP2/PyOP2
    (documentation at
    http://op2.github.io/PyOP2)

Postdoc positions in this area are available

- contact: me (lawrence.mitchell@imperial.ac.uk) or David Ham (david.ham@imperial.ac.uk)
Thanks

- Institutions
  - Imperial College London
  - Grantham Institute for climate change

- Funding
  - NERC (NE/K008951/1, NE/K006789/1, NE/G523512/1)
  - EPSRC (EP/L000407/1, EP/K008730/1, EP/I00677X/1)