Compiler technology for solving PDEs with performance portability

Paul H J Kelly
Group Leader, Software Performance Optimisation
Co-Director, Centre for Computational Methods in Science and Engineering
Department of Computing, Imperial College London

Joint work with:

David Ham (Imperial Computing/Maths/Grantham Inst for Climate Change)
Gerard Gorman, (Imperial Earth Science Engineering – Applied Modelling and Computation Group)
Mike Giles, Gihan Muralige, Istvan Reguly (Mathematical Inst, Oxford)
Doru Bercea, Fabio Luporini, Graham Markall, Lawrence Mitchell, Florian Rathgeber, George Rokos (Software Perf Opt Group, Imperial Computing)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)
Michelle Mills Strout, Chris Krieger, Cathie Olschanowsky (Colorado State University)
Carlo Bertolli (IBM Research)
Ram Ramanujam (Louisiana State University)
This talk is about the following idea:

- can we simultaneously raise the level at which programmers can reason about code,
- provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand?
This talk is about the following idea:

- can we simultaneously raise the level at which programmers can reason about code,
- provide the compiler with a model of the computation that enables it to generate faster code than you could reasonably write by hand?
Compilation is like skiing.

Analysis is not always the interesting part....
What we are doing...

**Projects**
- **PyOP2/OP2**
  - Unstructured-mesh stencils
- **Firedrake**
  - Finite-element assembly
- **PAMELA**
  - Dense SLAM – 3D vision
- **PRAgMaTIc**
  - Dynamic mesh adaptation
- **GiMMiK**
  - Small-matrix multiplication
- **TINTL**
  - Fourier interpolation

**Contexts**
- **Finite-volume CFD**
  - Real-time 3D scene understanding
- **Finite-element**
  - Adaptive-mesh CFD
- **Unsteady CFD - higher-order flux-reconstruction**
- **Ab-initio computational chemistry (ONETEP)**

**Technologies**
- **Vectorisation, parametric polyhedral tiling**
- **Tiling for unstructured-mesh stencils**
- **Lazy, data-driven compute-communicate**
- **Runtime code generation**
- **Multicore graph worklists**
- **Massive common sub-expressions**
- **Optimisation of composite transforms**

**Applications**
- Aeroengine turbo-machinery
- Weather and climate
- Domestic robotics, augmented reality
- Tidal turbines
- Formula-1, UAVs
- Solar energy, drug design

Targetting MPI, OpenMP, OpenCL, Dataflow/FPGA, from supercomputers to mobile, embedded and wearable.

Massive common sub-expressions
Vectorisation, parametric polyhedral tiling
Tiling for unstructured-mesh stencils
Lazy, data-driven compute-communicate
Runtime code generation
Multicore graph worklists
Massive common sub-expressions
Optimisation of composite transforms
<table>
<thead>
<tr>
<th>Projects</th>
<th>Contexts</th>
<th>Technologies</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyOP2/OP2</td>
<td>Finite-volume CFD</td>
<td>Vectorisation, parametric polyhedral tiling</td>
<td>Aeroengine turbo-machinery</td>
</tr>
<tr>
<td>Firedrake</td>
<td>Finite-element</td>
<td>Tiling for unstructured-mesh stencils</td>
<td>Weather and climate</td>
</tr>
<tr>
<td>PAMELA</td>
<td>Real-time 3D scene understanding</td>
<td>Lazy, data-driven compute-communicate</td>
<td>Tidal turbines</td>
</tr>
<tr>
<td>PRAgMatIc</td>
<td>Adaptive-mesh CFD</td>
<td>Runtime code generation</td>
<td>Domestic robotics, augmented reality</td>
</tr>
<tr>
<td>GiMMiK</td>
<td>Unsteady CFD - higher-order flux-reconstruction</td>
<td>Massive common sub-expressions</td>
<td>Formula-1, UAVs</td>
</tr>
<tr>
<td>TINTL</td>
<td>Ab-initio computational chemistry (ONETEP)</td>
<td>Optimisation of composite transforms</td>
<td>Solar energy, drug design</td>
</tr>
</tbody>
</table>

What we are doing...

Targetting MPI, OpenMP, OpenCL, Dataflow/FPGA, from supercomputers to mobile, embedded and wearable
OP2 and PyOP2: parallel loops over unstructured meshes

How well does it work?

Loop fusion and tiling for unstructured-meshes

Firedrake: a compiler for a higher-level DSL

COFFEE: a compiler for a lower-level DSL

This talk’s message:

Avoid analysis for transformational optimisation

Transform at the right level of abstraction

Design representations that get the abstraction right
- Unstructured mesh
- Sometimes adaptive (not in the rest of this talk)

- **OP2** is a C++ and Fortran library for parallel loops over the mesh implemented by source-to-source transformation
- **PyOP2** is a major extension implemented in Python using runtime code generation

- Generates highly-optimised CUDA, OpenMP and MPI code

- Key idea: parallel loop has access descriptor providing declarative specification of the data access

Example: mesh adaptation in AMCG’s Fluidity - http://amcg.ese.ic.ac.uk/index.php?title=Heated_Cylinder_Adapt_Example
- Unstructured mesh
- Sometimes adaptive (not in the rest of this talk)

**OP2** is a C++ and Fortran library for parallel loops over the mesh implemented by source-to-source transformation

**PyOP2** is an major extension implemented in Python using runtime code generation

- Generates highly-optimised CUDA, OpenMP and MPI code

- Key idea: parallel loop has access descriptor providing declarative specification of the data access
# PyOP2 – an active library for unstructured mesh computations

# declare sets, maps, and datasets

```python
nodes = op2.Set(nnode)
edges = op2.Set(nedge)

ppedge = op2.Map(edges, nodes, 2, pp)
```

```python
p_A = op2.Dat(edges, data=A)
p_r = op2.Dat(nodes, data=r)
p_u = op2.Dat(nodes, data=u)
p_du = op2.Dat(nodes, data=du)
```

# global variables and constants declarations

```python
alpha = op2.Const(1, data=1.0, np.float32)
beta = op2.Global(1, data=1.0, np.float32)
```

for iter in xrange(0, NITER):

```python
op2.par_loop(res, edges,
             p_A(op2.READ),
             p_u(op2.READ, pppedge[1]),
             p_du(op2.INC, pppedge[0]),
             beta(op2.READ))
```

```python
u_sum = op2.Global(1, data=0.0, np.float32)
u_max = op2.Global(1, data=0.0, np.float32)
```

```python
op2.par_loop(update, nodes,
              p_r(op2.READ),
              p_du(op2.RW),
              p_u(op2.INC),
              u_sum(op2.INC),
              u_max(op2.MAX))
```

Example – Jacobi solver

https://github.com/OP2/PyOP2/blob/master/demo/jacobi.py
PyOP2 – an active library for unstructured mesh computations

```python
for iter in xrange(0, NITER):
    op2.par_loop(res, edges,
                 p_A(op2.READ),
                 p_u(op2.READ, pedge[1]),
                 p_du(op2.INC, pedge[0]),
                 beta(op2.READ))

u_sum = op2.Global(1, data=0.0, np.float32)
u_max = op2.Global(1, data=0.0, np.float32)

op2.par_loop(update, nodes,
              p_r(op2.READ),
              p_du(op2.RW),
              p_u(op2.INC),
              u_sum(op2.INC),
              u_max(op2.MAX))
```

• In this simple example, the kernels are given as C strings
• In most of our work, the kernels are automatically generated
• And passed as ASTs

Example – Jacobi solver
Some key points about HYDRA:

- **Unmodified Fortran OP2 source code** exploits inter-node parallelism using MPI, and intra-node parallelism using OpenMP and CUDA.
- Application is a proprietary, full-scale, in-production fluids dynamics package.
- Developed by Rolls Royce plc and used for simulation of aeroplane engines.

(joint work with Mike Giles, Istvan Reguly, Gihan Mudalige at Oxford)

**“Performance portability”**

<table>
<thead>
<tr>
<th>System</th>
<th>CPU/Cores</th>
<th>GPU/Cores</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>HECToR (Cray XE6)</td>
<td>2x16-core AMD Opteron 6276 (Interlagos)2.3GHz</td>
<td>32GB</td>
<td>Jade (NVIDIA GPU Cluster)</td>
</tr>
<tr>
<td></td>
<td>Intel Xeon E5-1650 3.2GHz</td>
<td>5GB/GPU (ECC on)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Cray Gemini</td>
<td>FDR InfiniBand</td>
<td>CLE 3.1.29</td>
<td>Red Hat Linux Enterprise 6.3</td>
</tr>
<tr>
<td>Cray MPT 8.1.4</td>
<td>PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4</td>
<td>-O2 -xAVX</td>
<td></td>
</tr>
</tbody>
</table>

```bash
-O3 -h fp3 -h ipa5 -arch=sm_35 -use_fast_math
```
**Sparse tiling** on an unstructured mesh, for locality

- How can we fuse two loops, when there is a “halo” dependence?
- I.e., load a block of mesh and do the iterations of loop 1, then the iterations of loop 2, before moving to the next block.
- If we could, we could dramatically improve the memory access behaviour!

Strout, Lupormi et al, IPDPS'14
Tiling an unstructured mesh for locality

- Partition the iteration space of loop 1

Strout, Luporini et al, IPDPS'14
Partition the iteration space of loop 1
Colour the partitions

Tiling an unstructured mesh for locality

Strout, Luporini et al, IPDPS'14
Partition the iteration space of loop 1

- Colour the partitions
- Project the tiles, using the knowledge that colour n can use data produced by colour n-1
- Thus, the tile coloured #1 \textit{grows} where it meets colour #0
- And \textit{shrinks} where it meets colours #2 and #3
Partition the iteration space of loop 1

Colour the partitions

Project the tiles, using the knowledge that data produced by colour n-1

Thus, the tile coloured #1 grows where it meets colour #0 and shrinks where it meets colours #2 and #3

Inspector-executor: derive tasks and task graph from the mesh, at runtime

Strout, Luporini et al, IPDPS'14
Extreme results – OP2 loop fusion

Results on OP2 Airfoil

- Mesh size = 14M vertices
- # Loop chain = 2 loops
- No inspector/plans overhead

Speedup of Airfoil on Intel ManyCore (4-socket 10-core machine)

Threads
- Airfoil test problem
- Unstructured-mesh finite-volume
More realistic results – OP2 loop fusion

**Speedup of Airfoil on Sandy Bridge**

**Mesh size = 1.5M edges**

**# Loop chain = 6 loops**

**No inspector/plans overhead**

**Airfoil test problem**

**Unstructured-mesh finite-volume**
The finite element method in outline

Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector
Multilayered abstractions for FE

Local assembly:
- Specified using the FEniCS project’s DSL, UFL (the “Unified Form Language”)
- Computes local assembly matrix
- Key operation is evaluation of expressions over basis function representation of the element

Mesh traversal:
- OP2
- *Loops over the mesh*
- *Key is orchestration of data movement*

Solver:
- Interfaces to standard solvers, such as PetSc
A weak form of the shallow water equations

\[ \int_{\Omega} q \nabla \cdot u dV = - \int_{\Gamma_E} u \cdot n (q^+ - q^-) dS \]
\[ \int_{\Omega} v \cdot \nabla h dV = c^2 \int_{\Gamma_E} (h^+ - h^-) n \cdot v dS \]

can be represented in UFL as

**UFL source**

```python
V = FunctionSpace(mesh, 'Raviart-Thomas', 1)
H = FunctionSpace(mesh, 'DG', 0)
W = V*H
(v, q) = TestFunctions(W)
(u, h) = TrialFunctions(W)
M_u = inner(v, u)*dx
M_h = q*h*dx
Ct = -inner(avg(u), jump(q, n))*dS
C = c**2*adjoint(Ct)
F = f*inner(v, as_vector([-u[1], u[0]]))*dx
A = assemble(M_u+p_h+0.5*dt*(C-Ct+F))
A_r = M_u+p_h-0.5*dt*(C-Ct+F)
```

**Local assembly kernel**

```c
void Mass(double localTensor[3][3])
{
    const double qw[6] = { ... };
    const double CG1[3][6] = { ... };
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)
            for(int g = 0; g < 6; g++)
                localTensor[i][j] += CG1[i][g] * CG1[j][g] * qw[g];
}
```

**Parallel loop**
over all grid cells,
in unspecified order,
partitioned

**Unstructured grid**
defined by vertices,
edges and cells
**Firedrake: a finite-element framework**

- An alternative implementation of the FEniCS language
- Using PyOP2 as an intermediate representation of parallel loops
- All embedded in Python

- The FEniCS project’s UFL – DSL for finite element discretisation
- Compiler generates PyOP2 kernels and access descriptors
- Stencil DSL for *unstructured-mesh*
- Explicit *access descriptors* characterise access footprint of kernels
- Runtime code generation
The advection-diffusion problem:

Weak form:

\[ \int_\Omega q \frac{\partial T}{\partial t} \, dX = \int_{\partial\Omega} q(\nabla T - uT) \cdot n \, ds - \int_\Omega \nabla q \cdot \nabla T \, dX + \int_\Omega \nabla q \cdot uT \, dX \]

This is the entire specification for a solver for an advection-diffusion test problem.

Same model implemented in FEniCS/Dolfin, and also in Fluidity – hand-coded Fortran.

```fortran
  t=state.scalar_fields["Tracer"]            # Extract fields
  u=state.vector_fields["Velocity"]         # from Fluidity

  p=TrialFunction(t)                          # Setup test and
  q=TestFunction(t)                           # trial functions

  M=p*q*dx                                    # Mass matrix
  d=-dt*dfsVty*dot(grad(q),grad(p))*dx        # Diffusion term
  D=M-0.5*d                                   # Diffusion matrix

  adv = (q*t+dt*dot(grad(q),u)*t)*dx          # Advection RHS
  diff = action(M+0.5*d,t)                   # Diffusion RHS

  solve(M == adv, t)                         # Solve advection
  solve(D == diff, t)                        # Solve diffusion
```
Here we compare performance against two production codes solving the same problem on the same mesh:

- Fluidity: Fortran/C++
- DOLFIN: the FEniCS project’s implementation of UFL

Graph shows speedup over Fluidity on one core of a 12-core Westmere node
Phase separation of the two components of a binary fluid

Fourth-order parabolic time-dependent non-linear Cahn-Hilliard equation

GMRES solver with a fieldsplit preconditioner using a lower Schur complement factorisation

HYPRE Boomeramg

algebraic multigrid preconditioner

Example is in the demo suite

8M DOF mesh

Ten timesteps

Up to 1536 cores

Down to 5K DOFs per core

Running on ARCHER, a Cray XC30

Compute nodes contain two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) processors and 64GB of RAM in two 32GB NUMA regions.

Firedrake and PETSc were compiled with version 4.8.2 of the GNU Compilers and Cray MPICH2 6.3.1 with the asynchronous progress feature enabled was used for parallel runs. Generated code was compiled with the -O3 -mavx flags. The software revisions used were Firedrake revision c8ed154 from September 25 2014, PyOP2 revision f67fd39 from September 24 2014 with PETSc revision 42857b6 from August 21 2014 and DOLFIN revision 30bbd31 from August 22 2014 with PETSc revision d7ebadd from August 13 2014.

Generated code is compiled with -O3 -fno-tree-vectorize in Firedrake and -O3 -fast-math -march=native in DOLFIN

http://fenicsproject.org/documentation/dolfin/1.4.0/python/demo/documented/cahn-hilliard/ python/documentation.html
Firedrake – Scaling

Both Firedrake and Dolfin scale down to 10K DOFs/core
But Firedrake is much faster:
Better implementation of mixed spaces
Residuals and Jacobians are cached
Inlining and loop nest optimisations/vectorization
Solver is faster thanks to nested matrix handling of mixed spaces and Schur complement

http://fenicsproject.org/documentation/dolfin/1.4.0/python/demo/documentcd/cahn-hilliard/python/documentation.html
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
                     +((K1*X_D10[i][k]+K3*X_D01[i][k])*(K1*X_D10[i][j]+K3*X_D01[i][j]))+
                     +((K0*X_D10[i][k]+K2*X_D01[i][k])*(K0*X_D10[i][j]+K2*X_D01[i][j])))*
                     *det*W[i]);
}

Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.

The local assembly operation computes a small dense submatrix

Essentially computing (for example) integrals of flows across facets

These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE

LISTING 1:
Local assembly code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange p = 1 elements.

LISTING 2:
Local assembly code generated by Firedrake for a Burgers problem on a 3D tetrahedral mesh using Lagrange p = 1 elements.
void helmholtz(double A[3][4], double **coords) {
    #define ALIGN __attribute__((aligned(32)))
    // K, det = Compute Jacobian (coords)

    static const double W[3] ALIGN = {...}
    static const double X_D10[3][4] ALIGN = {{{...}}}
    static const double X_D01[3][4] ALIGN = {{{...}}}

    for (int i = 0; i<3; i++) {
        double LI_0[4] ALIGN;
        double LI_1[4] ALIGN;
        for (int r = 0; r<4; r++) {
            LI_0[r] = ((K1*X_D10[i][r])+(K3*X_D01[i][r]));
            LI_1[r] = ((K0*X_D10[i][r])+(K2*X_D01[i][r]));
        }
        for (int j = 0; j<3; j++)
            #pragma vector aligned
        for (int k = 0; k<4; k++)
            A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+LI_1[k]*LI_1[j])*det*W[i];
    }
}
void burgers(double A[12][12], double **coords, double **w) {
    // K, det = Compute Jacobian (coords)

    static const double W[5] = {...}
    static const double X1_D001[5][12] = {...}
    static const double X2_D001[5][12] = {...}
    //11 other basis functions definitions.

    ... 
    for (int i = 0; i<5; i++) {
        double F0 = 0.0;
        //10 other declarations (F1, F2,...)

    ... 
    for (int r = 0; r<12; r++) {
        F0 += (w[r][0]*X1_D100[i][r]);
        //10 analogous statements (F1, F2, ...)

    ...
    }
}

for (int j = 0; j<12; j++)
for (int k = 0; k<12; k++)
    A[j][k] += (..(K5*F9)+(K8*F10))*Y1[i][j]+
    +((K0*X1_D100[i][k])+(K3*X1_D010[i][k])+(K6*X1_D001[i][k]))*Y2[i][j])*F11+
    +((K2*X2_D100[i][k])+(K8*X2_D010[i][k]))*((K2*X2_D100[i][j])+(K8*X2_D001[i][j]))+
    +<roughly a hundred sum/muls go here>)*
    *det*W[i];
}
- Fairly serious, realistic example: static linear elasticity, $p=2$
tetrahedral mesh, 196608 elements
- Including both assembly time and solve time
- Single core of Intel Sandy Bridge
- Compared with Firedrake loop nest compiled with Intel’s icc compiler version 13.1
- At low $p$, matrix insertion overheads dominate assembly time
- At higher $p$, and with more coefficient functions ($f=2$), we get up to 1.47x overall application speedup
Conclusions: PyOP2 layer

The key idea in OP2/PyOP2 is access descriptors.

OP2’s access descriptors are declarative specifications of how each loop iteration is connected to the abstract mesh.

The kernels do not access the mesh.

The implementation is responsible for connecting the kernel to the data.

The implementation is free to select layout, stage data, schedule loops.

We can map from data to iterations.

What would a programming abstraction for data locality look like?
Conclusions: Firedrake layer

- Dramatically raised level of abstraction
- But we still can match or exceed hand-coded, in-production code
- Costs of abstraction are eliminated by dynamic generation of code specialised to context
- Domain-specific optimisations can yield big speedups over the best available general-purpose compilers

The real payoff lies in supporting the users in navigating freely to the best way to model their problem

How can the *barriers to adoption* of DSLs be overcome?
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

Code:

- [http://www.firedrakeproject.org/](http://www.firedrakeproject.org/)
- [http://op2.github.io/PyOP2/](http://op2.github.io/PyOP2/)
Welcome to PyOP2’s documentation!

Contents:

- Installing PyOP2
  - Quick start
  - Provisioning a virtual machine
  - Preparing the system
  - Dependencies
  - Building PyOP2
  - Setting up the environment
  - Testing your installation
  - Troubleshooting
- PyOP2 Concepts
  - Sets and mappings
  - Data
  - Parallel loops
- PyOP2 Kernels
  - Kernel API
  - Data layout
  - Local iteration spaces
- The PyOP2 Intermediate Representation
  - Using the Intermediate Representation
  - Achieving Performance Portability with the IR
  - Optimizing kernels on CPUs
  - How to select specific kernel optimizations
- PyOP2 Architecture
  - Multiple Backend Support
Firedrake is an automated system for the portable solution of partial differential equations using the finite element method (FEM). Firedrake enables users to employ a wide range of discretisations to an infinite variety of PDEs and employ either conventional CPUs or GPUs to obtain the solution.

Firedrake employs the Unified Form Language (UFL) and FEniCS Form Compiler (FFC) from the FEniCS Project and fields and meshes from Fluidity. The parallel execution of the FEM solver is accomplished by the PyOP2 system.

- The Firedrake team
  - Summer students 2013
- Obtaining Firedrake
  - PyOP2
  - Firedrake
The FEniCS project... The book

Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book (Lecture Notes in Computational Science and Engineering) [Hardcover]
Anders Logg (Editor), Kent-Andre Mardal (Editor), Garth Wells (Editor)

Price: £62.99 & this item Delivered FREE in the UK with Super Saver Delivery. See details and conditions

In stock but may require up to 2 additional days to deliver. Dispatched from and sold by Amazon. Gift-wrap available.

32 new from £48.11  8 used from £51.23