Delivering and generalising domain-specific program optimisations

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, Michael Lange (Imperial Earth Science Engineering – Applied Modelling and Computation Group)
Mike Giles, Gihan Mudalige, Istvan Reguly (Mathematical Inst, Oxford)
Doru Bercea, Fabio Luporini, Graham Markall, Lawrence Mitchell, Florian Rathgeber, Francis Russell, George Rokos,
Paul Colea (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)
Doru Thom Popovici, Franz Franchetti (CMU), Karl Wilkinson (Capetown), Chris–Kriton Skylaris (Southampton)
Compilation is like skiing...

- Analysis
  - Syntax
  - Types
  - Points-to
  - Class-hierarchy
  - Call-graph
  - Dependence
  - Shape
  - Polyhedra

- Points
  - Type Profiling

- Class
  - Hierarchies

- Control
  - Flow

- Data
  - Flow

- Instruction
  - Selection
  - Scheduling

- Storage
  - Layout

- Tiling

- Parallelisation
  - Loop nest ordering
  - Mapping

- Register allocation
  - Instruction selection/scheduling

Analysis is not always the interesting part....
It’s more fun the higher you start!
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?
### 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
- Ab-initio computational chemistry (ONETEP)
- Tiling for unstructured-mesh stencils
- Finite-volume
- Adaptive-mesh CFD
- Unsteady CFD - high-order flux-reconstruction
- Ab-initio computational chemistry (ONETEP)

### Contexts
- Finite-difference
- Finite-element
- Real-time 3D scene understanding
- Unsteady CFD - higher-order flux-reconstruction
- Fourier interpolation

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

### 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 HPC to mobile, embedded and wearable
What compilers can do
What stops the compiler from doing what it can do
What you might hope the compiler might do

Domain-specific optimisations
Getting the abstraction right
Delivering
Example:

```c
for (i=0; i<N; ++i) {
    points[i]->x += 1;
}
```

Can the iterations of this loop be executed in parallel?

No problem: each iteration is independent.
Example:

```c
for (i=0; i<N; ++i) {
    points[i]->x += 1;
}
```

Oh no: not all the iterations are independent!

- You want to re-use piece of code in different contexts
- Whether it’s parallel depends on context!
Example:
```c
for (i=0; i<N; ++i) {
    points[i]->x += 1;
}
```

Can the iterations of this loop be executed in parallel?

Sergio Almeida’s 1998 PhD thesis:
“Balloon types” ensure that each cell is reached only by its owner pointer – see also ownership in Rust
Points-to analysis

Goal: for each pointer variable (p, q, r, s), find the set of objects it might point to at runtime.

2006 PhD thesis work of David Pearce, (based on Andersen’94)

```
int *f(int *p) {
    return p;
}

int g() {
    int x, y, *p, *q, **r, **s;
    s=&p;
    if(...) p=&x;
    else p=&y;
    r=s;
    q=f(*r);
}
```

1. \( f_* \supseteq f_p \)
2. \( g_s \supseteq \{g_p\} \)
3. \( g_p \supseteq \{g_x\} \)
4. \( g_p \supseteq \{g_y\} \)
5. \( g_r \supseteq g_s \)
6. \( f_p \supseteq *g_r \)
7. \( g_q \supseteq f_* \)

- Variable \( s \) of function \( g \) might point to variable \( p \) of function \( g \)
- \( R \) might point to anything \( s \) might point to
- \( f \)'s \( p \) might point to anything \( r \) might point to
- \( q \) might point to anything \( f \) returns
Unstructured meshes require pointers/indirection because adjacency lists have to be represented explicitly.

A controlled form of pointers (actually a general graph)

**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
# 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)
```
PyOP2: “decoupled access-execute”

- Parallel loops, over sets (nodes, edges etc)
- Access descriptors specify how to pass data to and from the C kernel
- The kernel operates only on local data

for iter in xrange(0, NITER):
    u_sum = op2.Global(1, data=0.0, np.float32)
    u_max = op2.Global(1, data=0.0, np.float32)

    op2.par_loop(res, edges,
                 p_A(op2.READ),
                 p_u(op2.READ, p.edge[1]),
                 p_du(op2.INC, p.edge[0]),
                 beta(op2.READ))

    void res(float *A, float *u, float *du,
             const float *beta) {
        *du += (*beta) * (*A) * (*u);
    }

    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))

    void update(float *r, float *du, float *u,
                float *u_sum, float *u_max) {
        *u += *du + alpha * (*r);
        *du = 0.0f;
        *u_sum += (*u) * (*u);
        *u_max = *u_max > *u ? *u_max : *u;
    }
Code generation for indirect loops in OP2

- Supports diverse code generation schemes
- For MPI, OpenMP, GPU, and in prototype form for FPGA
- Key idea: inspector-executor

![Diagram of a graph with labeled nodes and edges. Partitions #53, #54, and #55 are shown. The edges are colored and directed. The diagram also includes a mention of cross-partition edges and a shared memory section.]

Eg for CUDA/OpenCL we can work with micro-partitions that we stage into GPU scratchpad memory, colour by colour.
Code generation for indirect loops in PyOP2

- For MPI we precompute partitions & haloes
- Derived from PyOP2 access descriptors, implemented using PETSc's DMPlex
- At partition boundaries, the entities (vertices, edges, cells) form layered halo region

Core: entities owned which can be processed without accessing halo data.

Owned: entities owned which access halo data when processed

Exec halo: off-processor entities which are redundantly executed over because they touch owned entities

Non-exec halo: off-processor entities which are not processed, but read when computing the exec halo
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>HECToR</th>
<th>Jade</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Cray XE6)</td>
<td>(NVIDIA GPU Cluster)</td>
</tr>
<tr>
<td>2×16-core AMD Opteron 6276 (Interlagos) 2.3GHz</td>
<td>2×Tesla K20m + Intel Xeon E5-1650 3.2GHz</td>
</tr>
<tr>
<td>64GB</td>
<td>5GB/GPU (ECC on)</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>Cray Gemini</td>
<td>FDR InfiniBand</td>
</tr>
<tr>
<td>CLE 3.129</td>
<td>Red Hat Linux Enterprise 6.3</td>
</tr>
<tr>
<td>Cray MPI 8.1.4</td>
<td>PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4</td>
</tr>
</tbody>
</table>

-O3 -h fp3 -h ipa5
-arch=smp_35 -use_fast_math
Sparse split 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, Luporini et al, IPDPS'14
To understand sparse split tiling, we need to first understand *split* tiling.

Consider a 1D stencil loop, iterated a number of times:

```c
for (t=0; t<N; ++t)
    for (i=1; i<M-1; ++i)
        U[t+1][i] = U[t][i-1] + U[t][i+1]
```

Tiling a *structured* mesh for locality
To understand sparse split tiling, we need to first understand *split* tiling.

Consider a 1D stencil loop iterated a number of times:

```c
for (t=0; t<N; ++t)
    for (i=1; i<M-1; ++i)
        U[t+1][i] = U[t][i-1] + U[t][i+1]
```

Lots of parallelism — but lots of data movement.
Sparse split tiling

Partition the iteration space of loop 1

Colour the partitions, execute the colours in order

Project the tiles, using the knowledge that colour \( n \) can use 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
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
OP2 loop fusion in practice

Mesh size = 1.5M edges
# Loop chain = 6 loops
No inspector/plans overhead

Airfoil test problem
Unstructured-mesh finite-volume

Breaking news: >30% performance improvement on full-scale Firedrake seismic inversion code

Speedup of Airfoil on Sandy Bridge

Intel Sandy Bridge (dual-socket 8-core Intel Xeon E5-2680 2.00Ghz, 20MB of shared L3 cache per socket); Intel icc-2013 (-O3, -xSSE4.2/-xAVX).
Where did the domain-specific advantage come from?

- OP2’s access descriptors provide precise dependence iteration-to-iteration information
  - Could easily be delivered in a lambda-based parallel loop framework
- We “know” that we will iterate many times over the same mesh – so it’s worth investing in an expensive “inspector-executor” scheme
- We capture chains of loops over the mesh
  - We could get our compiler to find adjacent loops
  - We could extend the OP2 API with “loop chains”
- What we actually do?
  - We delay evaluation of parallel loops
  - We build a chain (DAG) of parallel loops at runtime
  - We generate code at runtime for the traces that occur
The finite element method in outline:

\[
\text{do element} = 1,N \\
\text{assemble(element):}
\int_{\Omega} vL(u^\delta)\,dX = \int_{\Omega} vq\,dX.
\]

end do

\[Ax = b\]

Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector.
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:
- PyOP2
- Loops over the mesh
- Key is orchestration of data movement

Solver:
- Interfaces to standard solvers, such as PetSc
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 using runtime code generation

- 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

- Domain-specific loop optimizer
  - For finite-element assembly and similar loop nests
  - Vectorisation and flop-minimisation
**The advection-diffusion problem:**

$$\frac{\partial T}{\partial t} = D \nabla^2 T - \mathbf{u} \cdot \nabla T$$

**Weak form:**

$$\int_{\Omega} q \frac{\partial T}{\partial t} \, dX = \int_{\partial \Omega} q(\nabla T - \mathbf{u} T) \cdot \mathbf{n} \, ds - \int_{\Omega} \nabla q \cdot \nabla T \, dX + \int_{\Omega} \nabla q \cdot \mathbf{u} T \, dX$$

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**

```python
# Extract fields from Fluidity
M = p*q*dx
D = M - 0.5*d

# Advection RHS
adv = (q*t + dt*dot(grad(q), u)*t)*dx

# Diffusion RHS
diff = action(M + 0.5*d, t)

# Solve advection
solve(M == adv, t)
# Solve diffusion
solve(D == diff, t)
```

### Weak form:

$$\int_{\Omega} q \frac{\partial T}{\partial t} \, dX = \int_{\partial \Omega} q(\nabla T - \mathbf{u} T) \cdot \mathbf{n} \, ds - \int_{\Omega} \nabla q \cdot \nabla T \, dX + \int_{\Omega} \nabla q \cdot \mathbf{u} T \, dX$$
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

These results are preliminary and are presented for discussion purposes – see Rathgeber, Ham, Mitchell et al, http://arxiv.org/abs/1501.01809 for more systematic and up to date evaluation.

Graph shows speedup over Fluidity on one core of a 12-core Westmere node.
Where did the domain-specific advantage come from?

- UFL (the Unified Form Language, inherited from the FEniCS Project)
  - Delivers spectacular expressive power
  - Reduces scope for coding errors
  - Supports flexible exploration of different models, different PDEs, different solution schemes

- Building on PyOP2
  - Handles MPI, OpenMP, CUDA, OpenCL
  - Completely transparently

- PyOP2 uses runtime code generation
- So we don’t need to do static analysis
- So the layers above can freely exploit unlimited abstraction
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.

These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE.
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.
- These are combined to form a global system of simultaneous equations capturing the discretised conservation laws expressed by the PDE.
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]);
    }
}
Kernels are often a lot more complicated

```c
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_D001[i][k]))*((K2*X2_D100[i][j])...+(K8*X2_D001[i][j]))..+ + <roughly a hundred sum/muls go here>..)*
                    *det*W[i];
    }
}
```

- Local assembly code generated by Firedrake for a Burgers problem on a 3D tetrahedral mesh using Lagrange p = 1 elements
- Somewhat more complicated!
- Examples like this motivate more complex transformations
- Including loop fission
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
Where did the domain-specific advantage come from?

- Finite-element assembly kernels have complex structure
- With rich loop-invariant expression structure
- And simple dependence structure

COFFEE generates C code that we feed to the best available compiler

COFFEE’s transformations make this code run faster

COFFEE does not use any semantic information not available to the C compiler

But it does make better decisions

For the loops we’re interested in

For the linear operators arising in finite-element assembly we can show that it’s possible to \textit{minimise} the inner-loop flop count
Pointers lead to the compiler making conservative decisions

Idea: capture the key data structures at a higher level of abstraction

Let the tools “own the data” – and control its distribution

“inspector-executor” – take time to derive a schedule from the specific mesh at runtime

Your compiler doesn’t know things that you know

That you will iterate over the mesh many times without changing it

That the graph is easily-partitionable and colourable

Your compiler won’t do optimisations that we know are good for your code

Policy vs mechanism – good for your code might not be good in general

Runtime code generation is liberating

We do not try to do static analysis on client code

We encourage client code to use powerful abstractions
Where do DSO opportunities come from?

- Domain semantics (eg in SPIRAL)
- Domain expertise (eg we know that inspector-executor will pay off)
- Domain idiosyncracies (eg for GLICM)
- Transforming at the right representation
  - Eg fusing linear algebra ops instead of loops
- Data abstraction (eg AoS vs SoA)
  - Or whether to build the global system matrix (or instead to use a matrix-free or local-assembly scheme)

How can we engage with the application specialists to expose and automate domain-specific optimisations?
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?**
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?
Acknowledgements

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/)