Software abstractions for many-core software engineering

Paul H J Kelly
Group Leader, Software Performance Optimisation
Department of Computing
Imperial College London

Joint work with:

David Ham, Gerard Gorman, Florian Rathgeber (Imperial ESE/Grantham Inst for Climate Change Res)
Mike Giles, Gihan Mudalige (Mathematical Inst, Oxford)
Adam Betts, Carlo Bertolli, Graham Markall, Tiziano Santoro, George Rokos (Software Perf Opt Group, Imperial)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)
What we are doing:

- Moving meshes
- Mixed meshes

Roadmap: applications drive DSLs, delivering performance portability
Three slogans
- Generative, instead of transformative optimisation
- Get the abstraction right, to isolate numerical methods from mapping to hardware
- Build vertically, learn horizontally

Three stories
- Domain-specific active library examples
- General framework: access-execute descriptors
- The value of generative and DSL techniques
Parallelism breaks abstractions:
- Whether code should run in parallel depends on context
- How data and computation should be distributed across the machine depends on context

“Best-effort”, opportunistic parallelisation is almost useless:
- Robust software must robustly, predictably, exploit large-scale parallelism

How can we build robustly-efficient multicore software
While maintaining the abstractions that keep code clean, reusable and of long-term value?

It’s a software engineering problem
Active libraries and DSLs

- Domain-specific languages...
- Embedded DSLs
- Active libraries
  - Libraries that come with a mechanism to deliver library-specific optimisations

- Domain-specific “active” library encapsulates specialist performance expertise
- Each new platform requires new performance tuning effort
- So domain-specialists will be doing the performance tuning
- Our challenge is to support them

Applications

Active library

Exotic hardware

Visual effects
  - Finite element
  - Linear algebra
  - Game physics
  - Finite difference

GPU  Multicore  FPGA  Quantum?
Classical compilers have two halves

- Syntax
- Points-to
- Class-hierarchy
- Dependence
- Shape

- Program Dependence
- Parallelisation
- Tiling
- Storage layout
- Instruction selection/scheduling
- Register allocation
The right domain-specific language or active library can give us a free ride.
It turns out that analysis is not always the interesting part....
C, C++, C#, Java, Fortran

Code motion optimisations
Vectorisation and parallelisation of affine loops over arrays

Capture dependence and communication in programs over richer data structures

Specify application requirements, leaving implementation to select radically-different solution approaches
Encapsulating and delivering domain expertise

- Domain-specific languages & active libraries
  - Raise the level of abstraction
  - Capture a domain of variability
  - Encapsulate reuse of a body of code generation expertise/techniques
- Enable us to capture design space
- To match implementation choice to application context:
  - Target hardware
  - Problem instance
- This talk illustrates these ideas with some of our recent/current projects
// declare sets, maps, and datasets
op_set nodes = op_decl_set(nnodes);
op_set edges = op_decl_set(nedges);

op_map pedge1 = op_decl_map(edges, nodes, 1, mapData1);
op_map pedge2 = op_decl_map(edges, nodes, 1, mapData2);

op_dat p_A = op_decl_dat(edges, 1, A);
op_dat p_r = op_decl_dat(nodes, 1, r);
op_dat p_u = op_decl_dat(nodes, 1, u);
op_dat p_du = op_decl_dat(nodes, 1, du);

// global variables and constants declarations
float alpha[2] = { 1.0f, 1.0f };
op_decl_const( 2, alpha );

float u_sum, u_max, beta = 1.0f;

for ( int iter = 0; iter < NITER; iter++ )
{
    op_par_loop_4 ( res, edges,
                    op_arg_dat ( p_A, 0, NULL, OP_READ ),
                    op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
                    op_arg_dat ( p_du, 0, &pedge1, OP_INC ),
                    op_arg_gbl ( &beta, OP_READ ) );

    u_sum = 0.0f; u_max = 0.0f;

    op_par_loop_5 ( update, nodes,
                    op_arg_dat ( p_r, 0, NULL, OP_READ ),
                    op_arg_dat ( p_du, 0, NULL, OP_RW ),
                    op_arg_dat ( p_u, 0, NULL, OP_INC ),
                    op_arg_gbl ( &u_sum, OP_INC ),
                    op_arg_gbl ( &u_max, OP_MAX ) );
}
// declare sets, maps, and datasets
op_set nodes = op_decl_set( nnodes );
op_set edges = op_decl_set( nedges );

op_map pedge1 = op_decl_map (edges, nodes, 1, mapData1 );
op_map pedge2 = op_decl_map (edges, nodes, 1, mapData2 );

op_dat p_A = op_decl_dat (edges, 1, A );
op_dat p_r = op_decl_dat (nodes, 1, r );
op_dat p_u = op_decl_dat (nodes, 1, u );
op_dat p_du = op_decl_dat (nodes, 1, du );

// global variables and constants declarations
float alpha[2] = { 1.0f, 1.0f };
OP2 – a decoupled access-execute active library for unstructured mesh computations

Each parallel loop precisely characterises the data that will be accessed by each iteration

This allows staging into scratchpad memory

And gives us precise dependence information

In this example, the “res” kernel visits each edge

- reads edge data, A
- Reads beta (a global),
- Reads u belonging to the vertex pointed to by “edge2”
- Increments du belonging to the vertex pointed to by “edge1”

Example – Jacobi solver

float u_sum, u_max, beta = 1.0f;

for ( int iter = 0; iter < NITER; iter++ )
{
    op_par_loop_4 ( res, edges,
        op_arg_dat ( p_A, 0, NULL, OP_READ ),
        op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
        op_arg_dat ( p_du, 0, &pedge1, OP_INC ),
        op_arg_gbl ( &beta, OP_READ )
    );

    u_sum = 0.0f; u_max = 0.0f;

    op_par_loop_5 ( update, nodes,
        op_arg_dat ( p_r, 0, NULL, OP_READ ),
        op_arg_dat ( p_du, 0, NULL, OP_RW ),
        op_arg_dat ( p_u, 0, NULL, OP_INC ),
        op_arg_gbl ( &u_sum, OP_INC ),
        op_arg_gbl ( &u_max, OP_MAX )
    );
}
**Example – Jacobi solver**

In this example, the “res” kernel visits each edge
- reads edge data, A
- Reads beta (a global),
- Reads u belonging to the vertex pointed to by “edge2”
- Increments du belonging to the vertex pointed to by “edge1”

```c
inline void res(const float A[1], const float u[1],
    float du[1], const float beta[1])
{
    du[0] += beta[0]*A[0]*u[0];
}

inline void update(const float r[1], float du[1],
    float u[1], float u_sum[1], float u_max[1])
{
    u[0] += du[0] + alpha * r[0];
    du[0] = 0.0f;
    u_sum[0] += u[0]*u[0];
    u_max[0] = MAX(u_max[0],u[0]);
}
```

```c
float u_sum, u_max, beta = 1.0f;
for ( int iter = 0; iter < NITER; iter++ )
{
    op_par_loop_4 ( res, edges,
        op_arg_dat ( p_A, 0, NULL, OP_READ ),
        op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
        op_arg_dat ( p_du, 0, &pedge1, OP_INC  ),
        op_arg_gbl ( &beta, OP_READ )
    );
    u_sum = 0.0f; u_max = 0.0f;
    op_par_loop_5 ( update, nodes,
        op_arg_dat ( p_r, 0, NULL, OP_READ ),
        op_arg_dat ( p_du, 0, NULL, OP_RW ),
        op_arg_dat ( p_u, 0, NULL, OP_INC ),
        op_arg_gbl ( &u_sum, OP_INC ),
        op_arg_gbl ( &u_max, OP_MAX )
    );
}
```
Two key optimisations:

- Partitioning
- Colouring

Here we focus on GPU and multicore implementation

We also have MPI-level parallelisation

Exploring SSE/AVX

And FPGA
Two key optimisations:

- Partitioning
- Colouring
Two key optimisations:

- Partitioning
- Colouring

Elements of the edge set are coloured to avoid races due to concurrent updates to shared nodes.
Two key optimisations:

- **Partitioning**
- **Colouring**
  - At two levels
Example: non-linear 2D inviscid unstructured airfoil code, double precision (compute-light, data-heavy)

Two backends: OpenMP, CUDA (OpenCL coming)

For tough, unstructured problems like this GPUs can win, but you have to work at it

X86 also benefits from tiling; we are looking at how to enhance SSE/AVX exploitation
Combining MPI, OpenMP and CUDA

- non-linear 2D inviscid airfoil code
- 26M-edge unstructured mesh
- 1000 iterations
- Analytical model validated on up to 120 Westmere X5650 cores and 1920 HECToR (Cray XE6) cores

Unmodified C++ OP2 source code exploits inter-node parallelism using MPI, and intra-node parallelism using OpenMP and CUDA

(Preliminary results under review)
A higher-level DSL

Solving: \( \nabla^2 u = f \)

Weak form: \( \int \nabla v \cdot \nabla u \, dX = \int v f \, dX \) (Ignoring boundaries)

\[
\begin{align*}
\text{Psi} &= \text{state.scalar_fields}(\"psi\") \\
v &= \text{TestFunction}(\text{Psi}) \\
u &= \text{TrialFunction}(\text{Psi}) \\
f &= \text{Function}(\text{Psi}, \text{\texttt{\"\sin(x[0])+cos(x[1])\"}}) \\
A &= \text{dot(grad(v), grad(u))} \ast \text{dx} \\
\text{RHS} &= v \ast f \ast \text{dx} \\
\text{Solve} &= (\text{Psi}, A, \text{RHS})
\end{align*}
\]

UFL – Unified Form Language
(FEniCS project, http://fenicsproject.org/):
A domain-specific language for generating finite element discretisations of variational forms

Specify application requirements, leaving implementation to select radically-different solution approaches
The FE Method: computation overview

Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector.
Parallelising the global assembly leads to performance/correctness issues:

- Bisection search: uncoalesced accesses, warp divergence
- Contending writes: atomic operations, colouring

In some circumstances we can avoid building the global system matrix altogether

Goal: get the UFL compiler to pick the best option
The Local Matrix Approach

Why do we assemble $M$?

We need to solve $y = Mv$ where $M = A^T M^e A$

In the Local Matrix Approach we recompute this, instead of storing it:

$y = \left( A^T \left( M^e (A v) \right) \right)$

$b$ is explicitly required
Assemble it with an SpMV:

$b = A^T b^e$
Advection-Diffusion Equation:

\[
\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot \bar{\mathbf{u}} \cdot \nabla T
\]

Solved using a split scheme:
- Advection: Explicit RK4
- Diffusion: Implicit theta scheme

GPU code: expanded data layouts, with Addto or LMA

CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010] (Implemented within Fluidity)

Double Precision arithmetic

Simulation run for 200 timesteps

Simplified CFD test problem
Test Platforms

- **Nvidia 280GTX:**
  - 240 stream processors: 30 multiprocessors with 8 SMs each
  - 1GB RAM (4GB available in Tesla C1060)

- **Nvidia 480GTX:**
  - 480 stream processors: 15 multiprocessors with 32 SMs each
  - 1.5GB RAM (3GB available in Tesla C2050, 6GB in Tesla C2060)

- **AMD Radeon 5870:**
  - 1600 stream processors: 20 multiprocessors with 16 5-wide SIMD units
  - 1GB RAM (768MB max usable)

- **Intel Xeon E5620:**
  - 4 cores
  - 12GB RAM

**Software:**
- Ubuntu 10.04
- Intel Compiler 10.1 for Fortran (-O3 flag)
- NVIDIA CUDA SDK 3.1 for CUDA
- ATI Stream SDK 2.2 for OpenCL

**Linear Solver:**
- CPU: PETSc [Balay et al., 2010]
- CUDA Conjugate Gradient Solver [Markall & Kelly, 2009], ported to OpenCL
Advection-Diffusion Equation:
\[
\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot \bar{\mathbf{u}} \cdot \nabla T
\]

Solved using a split scheme:
- Advection: Explicit RK4
- Diffusion: Implicit theta scheme

GPU code: expanded data layouts, with Addto or LMA

CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010] (Implemented within Fluidity)

Double Precision arithmetic

Simulation run for 200 timesteps

On the 480GTX ("Fermi") GPU, local assembly is more than 10% slower than the addto algorithm (whether using atomics or with colouring to avoid concurrent updates)
- Advection-Diffusion Equation:
  \[
  \frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot \mathbf{\bar{u}} \cdot \nabla T
  \]

- Solved using a split scheme:
  - Advection: Explicit RK4
  - Diffusion: Implicit theta scheme

- GPU code: expanded data layouts, with Addto or LMA

- CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010]
  (Implemented within Fluidity)

- Double Precision arithmetic

- Simulation run for 200 timesteps

- On the quad-core Intel Westmere EP system, the local matrix approach is slower. Using Intel’s compiler, the baseline code (using addtos and without data expansion) is faster still
Throughput compared to CPU Implementation

Throughput of best GPU implementations relative to CPU (quad-core Westmere E5620) *(preliminary results, esp the AMD numbers)*

AMD 5870 and GTX480 kernel times very similar; older AMD drivers incurred overheads
The Local Matrix Approach is fastest on GPUs.

Global assembly with colouring is fastest on CPUs.

Expanded data layouts allow coalescing and higher performance on GPUs.

Accessing nodal data through indirection is better on CPU due to cache, lower memory bandwidth, and arithmetic throughput.
The balance between local- vs global-assembly depends on other factors. Eg tetrahedral vs hexahedral, Eg higher-order elements. Local vs Global assembly is not the only interesting option.

Relative execution time on CPU (dual quad Core2) - Helmholtz problem with Hex elements. 
With increasing order

(Cantwell et al, provisional results under review)
Mapping the design space – h/p

- Contrast: with tetrahedral elements
- Local is faster than global only for much higher-order
- Sum factorisation never wins

(Cantwell et al, provisional results under review)
Helmholtz problem using tetrahedral elements

What is the best combination of $h$ and $p$?

Depends on the solution accuracy required

Which, in turn determines whether to choose local vs global assembly

Blue dotted lines show runtime of optimal strategy; Red solid lines show $L_2$ error
Active libraries domain-specific languages and a unifying common framework

- Image processing/Visual effects active library
- Finite-difference ("stencil") DSL
- Unstructured-mesh DSL
- Molecular dynamics / discrete element DSL
- Other

AEcute: Kernels, iteration spaces, and access descriptors

Transformation and optimisation framework

- OpenCL

- nVidia Fermi
- Intel Larrabee
- AMD GPUs
- AVX
- Future platforms

Other back-ends, eg AVX intrinsics, FPGAs

A roadmap: taking a vertical view

General framework
Conclusions and Further Work

From these experiments:
- Algorithm choice makes a big difference in performance
- The best choice varies with the target hardware
- The best choice also varies with problem characteristics and accuracy objectives

We need to automate code generation
- So we can navigate the design space freely
- And pick the best implementation strategy for each context
Having your cake and eating it

- If we get this right:
  - Higher performance than you can reasonably achieve by hand
  - the DSL delivers reuse of expert techniques
  - Implements extremely aggressive optimisations
  - Performance portability
    - Isolate long-term value embodied in higher levels of the software from the optimisations needed for each platform
  - Raised level of abstraction
    - Promoting new levels of sophistication
    - Enabling flexibility
  - Domain-level correctness

Diagram:
- C/C++/Fortran
- DSL
- Reusable generator
- CUDA
- VHDL

Performance
Ease of use
Acknowledgements

Thanks to Lee Howes, Ben Gaster and Dongping Zhang at AMD

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