Generating high-performance multiplatform finite element solvers using the Manycore Form Compiler and OP2

Graham R. Markall, Florian Rathgeber, David A. Ham, Paul H. J. Kelly, Carlo Bertolli, Adam Betts
Imperial College London
Mike B. Giles, Gihan R. Mudalige
University of Oxford
Istvan Z. Reguly
Pazmany Peter Catholic University, Hungary
Lawrence Mitchell
University of Edinburgh
• How do we get performance portability for the finite element method?

• Using a form compiler with pluggable backend support
  – One backend: CUDA – NVidia GPUs

• Long term plan:
  – Target an *intermediate representation*
Manycore Form Compiler

- Compile-time code generation
  - Plans to move to runtime code generation
- Generates assembly and marshalling code
- Designed to support isoparametric elements
MCFC Pipeline

- Preprocessing: insert Jacobian and transformed gradient operators into forms
- Execution: Run in python interpreter, retrieve Form objects from namespace
- Form processing: compute_form_data()
- Partitioning: helps loop-nest generation
Preprocessing

• Handles coordinate transformation as part of the form using UFL primitives

```python
x = state.vector_fields['Coordinate']
J = Jacobian(x)
invJ = Inverse(J)
detJ = Determinant(J)
```

• Multiply each form by J

• Overloaded derivative operators, e.g.:

```python
def grad(u):
    return ufl.dot(invJ, ufl.grad(u))
```

• Code generation gives no special treatment to the Jacobian, its determinant or inverse
Loop nest generation

• Loops in typical assembly kernel:
  
  ```
  For (int i=0; i<3; ++i)
    For (int j=0; j<3; ++j)
      for (int q=0; q<6; ++q)
        for (int d=0; d<2; ++d)
  ```

• Inference of loop structure from preprocessed form:
  – Basis functions: use rank of form
  – Quadrature loop: Quadrature degree known
  – Dimension loops:
    • Find all the IndexSum indices
    • Recursively descend through form graph identifying maximal sub-graphs that share sets of indices
Partitioning example: $\int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \, dX$
Partitioning example: \( \int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \ dX \)
Partitioning example: \[ \int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \, dX \]
Partition code generation

• Once we know which loops to generate:
  – Generate an expression for each partition (subexpression)
  – Insert the subexpression into the loop nest depending on the indices it refers to
  – Traverse the topmost expression of the form, and generate an expression that combines subexpressions, and insert into loop nest
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        for (int q=0; q<6; ++q) {

        }
    }
}

\[ \int_{\Omega} \nabla v \cdot \nabla u + \lambda uv \, dX \]
for (int i=0; i<3; ++i) {  
    for (int j=0; j<3; ++j) {  
        LocalTensor[i,j] = 0.0;  
        for (int q=0; q<6; ++q) {  
            for (int d=0; d<2; ++d) {  
                
            }  
        }  
    }  
}
Code gen example: $\int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \, dX$

```c
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        LocalTensor[i,j] = 0.0;
        for (int q=0; q<6; ++q) {
            SubExpr0 = 0.0
            SubExpr1 = 0.0

            for (int d=0; d<2; ++d) {

            }

        }
    }
}
```
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        LocalTensor[i,j] = 0.0;
        for (int q=0; q<6; ++q) {
            SubExpr0 = 0.0
            SubExpr1 = 0.0
            SubExpr0 += arg[i,q]*arg[j,q]
            for (int d=0; d<2; ++d) {

            }
        }
    }
}
Code gen example: \[
\int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \, dX
\]

```c
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        LocalTensor[i,j] = 0.0;
        for (int q=0; q<6; ++q) {
            SubExpr0 = 0.0
            SubExpr1 = 0.0
            SubExpr0 += \text{arg}[i,q]*\text{arg}[j,q]
            for (int d=0; d<2; ++d) {
                SubExpr1 += \text{d_arg}[d,i,q]*\text{d_arg}[d,j,q]
            }
        }
    }
}
```
for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j) {
        LocalTensor[i,j] = 0.0;
        for (int q=0; q<6; ++q) {
            SubExpr0 = 0.0
            SubExpr1 = 0.0
            SubExpr0 += arg[i,q]*arg[j,q]
            for (int d=0; d<2; ++d) {
                SubExpr1 += d_arg[d,i,q]*d_arg[d,j,q]
            }
            LocalTensor[i,j] += SubExpr0 + SubExpr1
        }
    }
}
Benchmarking MCFC and DOLFIN

- Comparing and profiling assembly + solve of an advection-diffusion test case:

```python
Coefficient(FiniteElement("CG", "triangle", 1))
p=TrialFunction(t)
q=TestFunction(t)

diffusivity = 0.1

M=p*q*dx

adv_rhs = (q*t+dt*dot(grad(q),u)*t)*dx
t_adv = solve(M, adv_rhs)
d=-dt*diffusivity*dot(grad(q),grad(p))*dx

A=M-0.5*d
diff_rhs=action(M+0.5*d,t_adv)
tnew=solve(A,diff_rhs)
```
Experiment setup

- Term-split advection-diffusion equation
  - Advection: Euler timestepping
  - Diffusion: Implicit theta scheme
- Solver: CG with Jacobi preconditioning
  - Dolfin: PETSc
  - MCFC: From (Markall, 2009)
- CPU: 2 x 6 core Intel Xeon E5650 Westmere (HT off), 48GB RAM
- GPU Nvidia GTX480
- Mesh: 344128 unstructured elements, square domain. Run for 640 timesteps.
- Dolfin setup: Tensor representation, CPP opts on, form compiler opts off, MPI parallel
Adv-diff runtime

Linear scaling
MCFC CUDA Global Assembly
Dolfin

Time (s)

Number of Cores
Breakdown of solver runtime

Time (s)

- Diffusion Assembly
- Advection Assembly
- Diffusion Solve
- Advection Solve

MCFC

Dolfin (8 cores)
## Dolfin profile

<table>
<thead>
<tr>
<th>% Exec.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.8549</td>
<td>pair&lt;boost::unordered_detail::hash_iterator_base&lt;allocator&lt;unsigned ... &gt;::emplace()</td>
</tr>
<tr>
<td>11.9482</td>
<td>MatSetValues_MPIAIJ()</td>
</tr>
<tr>
<td>10.2417</td>
<td>malloc_consolidate</td>
</tr>
<tr>
<td>7.48235</td>
<td>_int_malloc</td>
</tr>
<tr>
<td>6.90363</td>
<td>dolfin::SparsityPattern::~SparsityPattern()</td>
</tr>
<tr>
<td>2.60801</td>
<td>dolfin::UFC::update()</td>
</tr>
<tr>
<td>2.48799</td>
<td>MatMult_SeqAIJ()</td>
</tr>
<tr>
<td>2.48758</td>
<td>ffc_form_d2c601cd1b0e28542a53997b6972359545bb30cc_cell_integral_0_0::tabulate_tensor()</td>
</tr>
<tr>
<td>2.3168</td>
<td>/usr/lib/openmpi/lib/libopen-pal.so.0.0.0</td>
</tr>
<tr>
<td>2.22407</td>
<td>boost::unordered_detail::hash_table&lt;boost::unordered_detail::set&lt;boost::hash&lt;... &gt;::rehash_impl()</td>
</tr>
<tr>
<td>1.9389</td>
<td>dolfin::MeshEntity::entities()</td>
</tr>
<tr>
<td>1.89775</td>
<td>_int_free</td>
</tr>
<tr>
<td>1.83794</td>
<td>free</td>
</tr>
<tr>
<td>1.71037</td>
<td>malloc</td>
</tr>
<tr>
<td>1.5123</td>
<td>/usr/lib/openmpi/lib/openmpi/mca_btl_sm.so</td>
</tr>
<tr>
<td>1.47677</td>
<td>/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.15</td>
</tr>
<tr>
<td>1.47279</td>
<td>poll</td>
</tr>
<tr>
<td>1.42863</td>
<td>ffc_form_958612b38a9044a3a64374d9d4be0681810f6bd8_cell_integral_0_0::tabulate_tensor()</td>
</tr>
<tr>
<td>1.18228</td>
<td>dolfin::SparsityPattern::insert()</td>
</tr>
<tr>
<td>1.13536</td>
<td>ffc_form_ba88085bc231bf16ec1c084f12b9c723279414f1_cell_integral_0_0::tabulate_tensor()</td>
</tr>
<tr>
<td>1.08694</td>
<td>ffc_form_23b22f19865ca4de78804edcf2815d350d5a55a3_cell_integral_0_0::tabulate_tensor()</td>
</tr>
<tr>
<td>0.983646</td>
<td>dolfin::GenericFunction::evaluate()</td>
</tr>
<tr>
<td>0.95484</td>
<td>dolfin::Function::restrict()</td>
</tr>
<tr>
<td>0.869109</td>
<td>VecSetValues_MPI()</td>
</tr>
<tr>
<td>% Exec.</td>
<td>Kernel</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>28.7</td>
<td>Matrix addto</td>
</tr>
<tr>
<td>14.9</td>
<td>Diffusion matrix local assembly</td>
</tr>
<tr>
<td>7.1</td>
<td>Vector addto</td>
</tr>
<tr>
<td>4.1</td>
<td>Diffusion RHS</td>
</tr>
<tr>
<td>2.1</td>
<td>Advection RHS</td>
</tr>
<tr>
<td>0.5</td>
<td>Mass matrix local assembly</td>
</tr>
<tr>
<td></td>
<td>Solver kernels</td>
</tr>
</tbody>
</table>
Thoughts

• Targeting the hardware directly allows for efficient implementations to be generated
• The MCFC CUDA backend embodies form-specific and hardware specific knowledge
• We need to target a performance portable intermediate representation
Layers manage complexity. Each layer of the IR:
- New optimisations introduced that are not possible in the higher layers
- With less complexity than the lower layers

**OP2: Unstructured mesh**

**Domain-specific language (DSL)**

- Unified Form Language (Form Compiler)
- Local assembly
  - Quadrature vs. Tensor
  - FErari optimisations
- Global assembly
  - Matrix format
  - Assembly algorithm
  - Data structures

**Backend-specific**
- "Classic" opts.

**Large parallel clusters using MPI**

**Multicore CPUs using OpenMP, SSE, AVX**

**GPUs using CUDA and OpenCL**

**Streaming dataflow using FPGAs**
Why OP2 for MCFC?

- Isolates a *kernel* that performs an operation for *every* mesh component – (Local Assembly)
- The job of OP2 is to control all code necessary to apply the kernel, fast
- Pushing all the OpenMP, MPI, OpenCL, CUDA, AVX issues into the OP2 compiler.
- Abstracts away the matrix representation so OP2 controls whether (and how/when) the matrix is assembled.
Helmholtz solver runtime breakdown

- **Solve**
- **Assemble**
- **Build Sparsity**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Solve</th>
<th>Assemble</th>
<th>Build Sparsity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCFC-OP2-Seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCFC-OP2-CUDA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dolfin</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• High performance implementations are obtained by flattening out abstractions
• Flattening abstractions increases complexity – we need to combat this with a new, appropriate abstraction
• This greatly reduces the implementation space for the form compiler to work with
• Whilst still allowing performance portability
• MCFC OP2 implementation: ongoing
Spare slides
MCFC Compile/run flow

Fluidity Markup File

SPUD

Options Tree

Executable

Simulation Output

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code

MCFC

UFL Code, State (Fields, Elements)

OP2 Code

OP2 Translator

NVIDIA Compiler

CUDA Code

Simulation Executable

Object Code

Linker

Fluidity Runtime Code
OP2 Matrix support

• *Matrix support* follows from Iteration Spaces:
  – What is the mapping between threads and elements? Example, on GPUs:
    – For low-order, one thread per element
    – For higher-order, one thread block per element
• OP2 extends iteration spaces to the matrix indices
• OP2 abstracts them completely from the user – they're inherently temporary data types
• There's no concept of getting the matrix back from op2.
void mass(float *A, float *x[2], int i, int j)
{
    int q;
    float J[2][2];
    float detJ;
    const float w[3]= {0.166667, 0.166667, 0.166667};
    const float CG1[3][3] = {{0.666667, 0.166667, 0.166667},
                              {0.166667, 0.666667, 0.166667},
                              {0.166667, 0.166667, 0.666667}};

    J[0][0] = x[1][0] - x[0][0];
    J[0][1] = x[2][0] - x[0][0];
    J[1][0] = x[1][1] - x[0][1];
    J[1][1] = x[2][1] - x[0][1];

    detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

    for ( q = 0; q < 3; q++ )
        *A += CG1[i][q] * CG1[j][q] * detJ * w[q];
void mass(float *A, float *x[2], int i, int j)

op_par_loop(mass, op_iteration_space(elements, 3, 3),

op_arg_mat(mat, op_i(1), elem_node, op_i(2), elem_node, OP_INC),

op_arg_dat(xn, OP_ALL, elem_node, OP_READ));
The OP2 abstraction

• The mesh is represented in a general manner as a graph. Primitives:
  – Sets (e.g. cells, vertices, edges)
  – mappings (e.g. from cells to vertices)
  – datasets (e.g. coefficients)

• No mesh entity requires special treatment
• Cells, vertices, etc are entities of different arity
The OP2 abstraction

• Parallel loops specify:
  – A *kernel*
  – An *Iteration space*: A set
  – An *Access Descriptor*: Datasets to pass to the kernel, and the mappings through which they’re accessed

• OP2 Runtime handles application of the kernel at each point in the iteration space, feeding the data specified in the access descriptor