Status of effective translation of complicated forms in FEniCS

The UFLACS project

Martin Sandve Alnæs

Center for Biomedical Computing,
Simula Research Laboratory,
Oslo, Norway

March 18th, 2013
FEniCS’13
The uflacs project - what is working, what is not

Preliminary benchmark results

Short overview of algorithms
A key feature in FEniCS is the translation from symbolic equations to efficient low level code

- The symbolic equations are written in UFL code
- The translation is performed by the FEniCS Form Compiler
- FFC fails when the equations reach a certain complexity
- Uflacs is a project with new compiler algorithms to fix this
Uflacs can be installed today and used as a third representation in ffc

bzar branch lp:uflacs; cd uflacs
python setup.py install --prefix=/your/fenics/path

```
from dolfin import *
# Use uflacs for everything:
parameters["form_compiler"]["representation"] = "uflacs"

# Or use uflacs for only this form:
p = {
    "representation":"uflacs"
}   
A = assemble(a, form_compiler_parameters=p)
```

```
ffc -r uflacs -l dolfin ffc/demo/HyperElasticity.ufl
g++ -c HyperElasticity.h
```
To reach full feature completeness with uflacs, there are a bunch of (mostly small) fixes left

- Integrals: $dx$, $ds$; $dS$, $dP$
- Expressions: almost everything; conditionals, jump, avg, higher order derivatives
- Geometry: $x$ on cell, circumradius, facet normal, ...; $x$ on facet
- Elements: full mixed element support; non-standard element mappings, quadrature elements

(This is obviously not a complete list).
Topics

The uflacs project - what is working, what is not

Preliminary benchmark results

Short overview of algorithms
For a form compiler, there are three kinds of performance, all important

- Code generation time
- C++ compile time
- Assembly time

NB! The performance measurements presented next are done quickly as a reality check, this is still work in progress.
A basic hyperelastic model (see ffc demo)

```
# Copyright (C) 2009 Harish Narayanan

element = VectorElement("Lagrange", tetrahedron, 1)

v = TestFunction(element)  # Test function
du = TrialFunction(element)  # Incremental displacement
u = Coefficient(element)    # Previous displacement
B = Coefficient(element)    # Body force per unit mass
T = Coefficient(element)    # Traction force on boundary
F = Identity(3) + grad(u)  # Deformation gradient
C = F.T*F                  # Right Cauchy-Green tensor
E = variable((C-Identity(3))/2)  # Euler-Lagrange strain tensor
mu = Constant(tetrahedron)  # Lame's constants
lam = Constant(tetrahedron)

psi = lam/2*(tr(E)**2) + mu*tr(E*E)  # Strain energy function
S = diff(psi, E)  # Second Piola-Kirchhoff stress tensor

# The variational problem corresponding to hyperelasticity
L = inner(F*S, grad(v))*dx - inner(B, v)*dx - inner(T, v)*ds
a = derivative(L, u, du)
```
Comparing uflacs to quadrature representation for HyperElasticity.ufl – time to build

All numbers provided by ffc bench suite:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Generate</th>
<th>Compile</th>
<th>Compile -O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>0.8 s</td>
<td>1.0 s</td>
<td>3 s</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>12.9 s</td>
<td>1.6 s</td>
<td>5.1 s</td>
</tr>
</tbody>
</table>
Comparing uflacs to quadrature representation for HyperElasticity.ufl – time to compute (1)

All numbers provided by ffc bench suite:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Generate</th>
<th>Compile</th>
<th>Compile -O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>0.8 s</td>
<td>1.0 s</td>
<td>3 s</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>12.9 s</td>
<td>1.6 s</td>
<td>5.1 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime without -O2</th>
<th>a</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>11.91 μs</td>
<td>4.25 μs</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>9.37 μs</td>
<td>8.62 μs</td>
</tr>
</tbody>
</table>
Comparing uflacs to quadrature representation for HyperElasticity.ufl – time to compute (2)

All numbers provided by ffc bench suite:

<table>
<thead>
<tr>
<th>Representation</th>
<th>Generate</th>
<th>Compile</th>
<th>Compile -O2</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>0.8 s</td>
<td>1.0 s</td>
<td>3 s</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>12.9 s</td>
<td>1.6 s</td>
<td>5.1 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime without -O2</th>
<th>a</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>11.91 µs</td>
<td>4.25 µs</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>9.37 µs</td>
<td>8.62 µs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runtime with -O2</th>
<th>a</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>2.72 µs</td>
<td>1.10 µs</td>
</tr>
<tr>
<td>quadrature -O</td>
<td>2.65 µs</td>
<td>2.65 µs</td>
</tr>
</tbody>
</table>
Uflacs provides twice as fast assembly in dolfin hyperelasticity demo

<table>
<thead>
<tr>
<th>Assemble cells</th>
<th>Average time</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>0.27 s</td>
</tr>
<tr>
<td>quadrature</td>
<td>0.55 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assemble facets</th>
<th>Average time</th>
</tr>
</thead>
<tbody>
<tr>
<td>uflacs</td>
<td>0.02295</td>
</tr>
<tr>
<td>quadrature</td>
<td>0.02252</td>
</tr>
</tbody>
</table>

Numbers provided by timings().
Uflacs enables new applications in FEniCS: Here large deformation of a left ventricle with anisotropic hyperelastic material
An excerpt of a Fung type anisotropic hyperelasticity model – previously not feasible in FEniCS

```
# Identity matrix and global deformation gradient
F_glob = I + grad(u)
F = variable(R.T*F_glob*R)
E = 0.5*(F.T*F - I)
J = det(F)

# Fung-type material law
f=0; s=1; n=2
W = (bff*E[f,f]**2 + bxx*(E[n,n]**2 + E[s,s]**2 + E[n,s]**2) + bfx*(E[f,n]**2 + E[n,f]**2 + E[f,s]**2 + E[s,f]**2))
psi = 0.5*K*(exp(W) - 1) + Ccompr*(J*ln(J) - J + 1)
P = R*diff(psi, F)*R.T # First Piola-Kirchoff stress tensor

# Neumann boundary condition
sigma = Constant(-0.02)
T = dot(det(F_glob)*sigma*inv(F_glob.T), N)
```
Time to jit and assemble matrix for Poisson compared to Fung type anisotropic hyperelasticity

<table>
<thead>
<tr>
<th>assemble(a)</th>
<th>tensor/P</th>
<th>quadr/P</th>
<th>uflacs/P</th>
<th>uflacs/Fung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clean cache</td>
<td>2.367 s</td>
<td>2.506 s</td>
<td>2.452 s</td>
<td>7.077 s</td>
</tr>
<tr>
<td>Memory cache</td>
<td>0.045 s</td>
<td>0.068 s</td>
<td>0.218 s</td>
<td>0.568 s</td>
</tr>
<tr>
<td>Disk cache</td>
<td>0.049 s</td>
<td>0.067 s</td>
<td>0.216 s</td>
<td>1.644 s</td>
</tr>
<tr>
<td>Memory cache</td>
<td>0.045 s</td>
<td>0.062 s</td>
<td>0.213 s</td>
<td>0.567 s</td>
</tr>
</tbody>
</table>
Topics

The uflacs project - what is working, what is not

Preliminary benchmark results

Short overview of algorithms
UFL represents symbolic expressions as a Directed Acyclic Graph (DAG)

- Each node is represented by a subclass of Terminal or Operator
- Each node can be tensor valued
- Some operators represent computation (e.g. addition)
- Other operators represent only reshaping (e.g. indexing)
UFLACS was designed for tensor intensive equations – that make heavy use of tensor algebra features in UFL

- Algorithms produce in a lot of symbolic patterns similar to indexing → scalar operators → indexed-to-tensor
- Operations such as A[i,j,k], as_tensor(A[i,j,k],(k,i,j)), and A.T should not contribute to computations but increase symbolic complexity
- Uflacs algorithms were designed with this in mind
The initial stages of the uflacs compiler algorithm

- Translate the DAG from node-based to list-based representation
- Apply value numbering of each scalar subexpression component involving a computation
- Value numbering “falls through” reshaping type operators
After the initial stages, the expression has been translated to a list of scalar expressions

- Each subexpression is either
  - a scalar operator performing some computation, or
  - a *modified terminal*

- *Modified terminals* are terminals with eventual grad, restriction, and indexed operators applied

- A modified terminal represents a scalar expression that uflacs does not know how to compute (*needs geometry or elements*)
In the intermediate stages, dependencies are represented and analysed using integer arrays

- Easy with array based DAG storage with scalar nodes
- Edges are therefore efficient to invert and count
- Only modified terminals that are referenced by operator nodes are stored
- Edge arrays are used to e.g.
  - Decide loop placement of subexpressions
  - Prioritize intermediate variable storage of subexpressions
  - (Quite crude algorithms at this stage)
In the code generation stage, a generic code generator delegates modified terminals to a backend

- A generic compiler routine in uflacs produces C(++) code with backend-specific code inserted on demand.
- An ffc backend in uflacs generates code to compute modified terminals based on tables of element basis function values passed from FFC.
- A dolfin backend in uflacs generates a dolfin::Expression subclass, including code to evaluate a GenericFunction member inside the Expression::eval implementation.
Current state of ffc-uflacs project relations (it’s not as messy as it may sound...)

- ffc uses ffc.uflacsrepr to generate `tabulate_tensor`
- ffc.uflacsrepr delegates most of the work to uflacs.backends.ffc
- uflacs.backends.ffc uses the generic uflacs.algorithms.compiler to do most of the work, passing it callbacks to generate code for computing modified terminals (geometry and functions)
Questions?

- Try uflacs on your forms at the “Ask the developer” session later today!
- Report bugs to http://bugs.launchpad.net/uflacs
- If you have a form that still takes long to build, send it to me and I can use it for profiling later.
- martinal@simula.no