Exploring Symbolic Manipulation and other Code Generation Techniques for Finite Element Local Assembly

FEniCS'11 Presentation

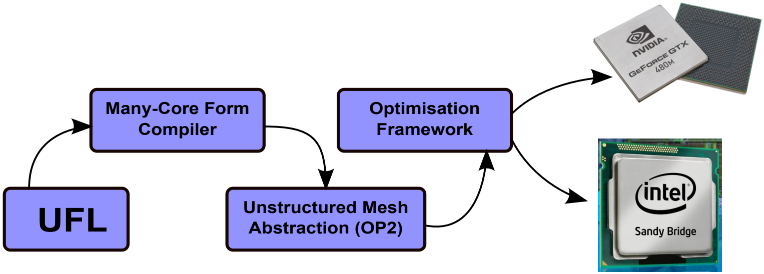
Francis Russell

Imperial College London
Department of Computing

5/11/2011

Software Performance Optimisation Group

- Member of Software Performance Optimisation (SPO) group at Imperial College London.
- Other investigators are Mike Giles (Oxford), David Ham (Imperial College Earth Science and Engineering) and Michael Fagan (University of Hull Medical and Biological Engineering).
- Also includes Graham Markall and Florian Rathgeber who presented their work at FEniCS'10.



Local Assembly

- Traditional approach has been to use quadrature.
- Development of the Unified Form Language (UFL) and the FEniCS Form Compiler (FFC) have facilitated the exploration of other implementations, especially those too complex to hand-implement.
- The tensor contraction implementation of local assembly[1] (in particular the topological optimisations) is a prime example of this.
- Ølgaard and Wells have analysed[2] the differing performance characteristics of quadrature and tensor implementations.
- Can we use symbolic algebra techniques to find novel implementation choices that outperform quadrature and tensor contraction implementations?

[1] R. Kirby, A. Logg, L. Ridgeway Scott, A. Terrel, "Topological Optimization of the Evaluation of Finite Element Matrices", 2006.

[2] K. Ølgaard, G. Wells, "Optimizations for Quadrature Representations of Finite Element Tensors through Automated Code Generation", 2010.

Symbolic Techniques

These techniques are not new:

- These techniques have been investigated as early as 1984 by Wang during development of the FINGER system.
- More recently, investigated by Alnæs and Mardall in the SyFi Form Compiler (a FEniCS sub-project).

Typically:

- Treats each entry of the local assembly matrix as an independent expression.
- Using quadrature or symbolic integration, computes the integral of each expression over a general cell.
- Applies common sub-expression elimination techniques to reduce computation cost by exploiting inter and intra expression redundancy.

What's new?

- Apply recent research on efficiently evaluating sets of multivariate polynomial expressions.
- We scale these techniques to some (relatively) large problem sizes.
- We extend this research to take account of the numerical relationships between expressions to improve the evaluation strategies that can be found.
- We compare operation counts of code generated by our library (EXCAFÉ) against FFC generated quadrature and tensor contraction implementations over a range of forms.
- We compare how effectively the Intel C++ Compiler and the GNU C++ Compiler can optimise these implementations.
- We look at the numerical accuracy effects of the tensor contraction topological optimisations as well.

Quadrature

Take the Laplace operator:

$$a(u, v) = \int_{\Omega} \nabla u(x) \cdot \nabla v(x)\, dx$$

Evaluation of the $p \times q$ local assembly matrix by quadrature for some cell k involves evaluating a weighted sum at Q points over the element volume:

$$M_{qp}^{k} = \sum_{i=0}^{Q-1} w_i (\nabla \chi^k)^{-1} \cdot \nabla \phi_p \cdot (\nabla \chi^k)^{-1} \cdot \nabla \psi_q |J(\chi^k)|$$

χ^k is the local-to-global coordinate mapping for cell k.
Q is usually determined by the polynomial order of the form.

Tensor Contraction

Tensor contraction representation involves representing the local assembly matrix as a contraction of a geometry-independent reference tensor (A^0) and a geometry-dependent tensor (G_k).

$$M^k = A^0 : G_k$$

For the Laplacian example, these can be defined as follows:

$$A^0_{qp\alpha\beta} = \int_{\Omega_{st}} \frac{\partial \phi_p}{\partial \xi_\alpha} \frac{\partial \psi_q}{\partial \xi_\beta} \, d\xi$$

$$G_k^{\alpha\beta} = |J(\chi^k)| \sum_{\gamma=0}^{d} \frac{\partial \xi_\alpha}{\partial x_\gamma} \frac{\partial \xi_\beta}{\partial x_\gamma}$$

The cost of performing the tensor contraction can be reduced through topological analysis of the reference tensor.

Quadrature versus Tensor Contraction

- Ølgaard and Wells have compared of the evaluation cost of different classes of bilinear forms using quadrature and tensor contraction based implementations.
- Tensor contraction performs better with high-order basis functions whereas quadrature performs better with forms that contain large numbers of functions and/or derivatives.
- The operation count ratio for quadrature versus tensor contraction based assembly can be anything from 0.01 to 350, for extreme (but not unrealistic) cases.
- Both quadrature and tensor contraction can be considered particular strategies for evaluating and reusing certain sub-expressions.
- The symbolic approach makes it trivial to support arbitrary sub-expressions.
- Key to efficient code generation is to exploit domain-specific knowledge about variational forms and basis functions.

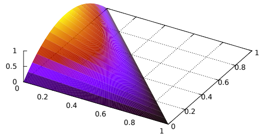
Characteristics of the Symbolic Manipulation Approach

- The integral can be evaluated at code-generation time (either analytically or symbolically) for linear elements, making the cost of assembly independent of the degree of the basis functions.
- Symbolic integration can be extremely computationally expensive to perform on large expressions, even when using optimised computer algebra systems such as Maxima.
- Treating each expression individually makes it impossible to generate the loop structures that are used in quadrature and tensor contraction based implementations.
- We note that the tensor contraction topological optimisations also destroy these loop structures.

What We Do

- We manipulate representations of variational forms and basis functions at code-generation time.

$$\int_{\Omega} \alpha \nabla\phi \cdot \nabla\psi dX$$



Bilinear forms Basis functions

- The local-to-global geometry transformation can be represented symbolically.
- We can apply differential operators such as *grad* and *div* to symbolic representations of our basis functions.
- We symbolically integrate these expressions over the reference cell.

Symbolically factorising local assembly

- After symbolic integration, we have independent multivariate rational expressions for each entry of the local assembly matrix.
- Generating efficient code from these requires identifying and reusing certain computations.
- Standard compiler CSE passes neither have the freedom nor the capacity to take advantage of the numerical relationships we wish to exploit.
- In particular, we want to be able to perform optimisations that take advantage of the *distributivity of multiplication over addition*.
- We have extended existing work by Hosangadi et al.[3] on optimising evaluation of sets of multivariate polynomials as part of our local assembly code generator.

[3]A. Hosanagadi, F. Fallah, R. Kastner, "Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination", 2006.

The Hosangadi et al. Algorithm

- Handles extracting common subexpressions across multiple independent polynomial expressions.
- For each polynomial expression F, forms factorisations of the form $C * F_1 + F_2$ where C is a monomial and F_1 and F_2 are polynomials.
- Example: $e_1 = x^3 + 2x^2y + y$

$$\begin{aligned} &= 1(x^3 + 2x^2y + y) \qquad (1a)\\ &= y(2x^2 + 1) + x^3 \qquad (1b)\\ &= x^2(x + 2y) + y \qquad (1c)\end{aligned}$$

- Example: $e_2 = x^3 + 2x^2 + 1$

$$\begin{aligned} &= 1(x^3 + 2x^2 + 1) \qquad (2a)\\ &= x^2(x + 2) + 1 \qquad (2b)\end{aligned}$$

- The search space of possible new subexpressions is expressed as a matrix.

The Factorisation Matrix

Rows correspond to different factorisations of each expression.
Columns correspond to terms in those factorisations.
Subscripts denote term numberings.

$$e_1 = {x^3}_{(1)} + 2x^2y_{(2)} + y_{(3)}$$
$$e_2 = {x^3}_{(4)} + {2x^2}_{(5)} + 1_{(6)}$$

$$\begin{array}{cc|cccccccc}
 & & 1 & 2 & 2x^2 & 2x^2y & 2y & x & x^3 & y \\
\hline
(e_1) & 1 & 0 & 0 & 0 & 1_{(2)} & 0 & 0 & 1_{(1)} & 1_{(3)} \\
(e_1) & y & 1_{(3)} & 0 & 1_{(2)} & 0 & 0 & 0 & 0 & 0 \\
(e_1) & x^2 & 0 & 0 & 0 & 0 & 1_{(2)} & 1_{(1)} & 0 & 0 \\
(e_2) & 1 & 1_{(6)} & 0 & 1_{(5)} & 0 & 0 & 0 & 1_{(4)} & 0 \\
(e_2) & x^2 & 0 & 1_{(5)} & 0 & 0 & 0 & 1_{(4)} & 0 & 0
\end{array}$$

The Factorisation Matrix

Factorisations correspond to a subset of rows and columns in which every entry is equal to one.

$$
\begin{aligned}
f &= 2x^2 + 1 \\
e_1 &= x^3{}_{(1)} + 2x^2y_{(2)} + y_{(3)} \\
e_2 &= x^3{}_{(4)} + 2x^2{}_{(5)} + 1_{(6)}
\end{aligned}
$$

$$
\begin{array}{cc|cccccccc}
 & & 1 & 2 & 2x^2 & 2x^2y & 2y & x & x^3 & y \\
\hline
(e_1) & 1 & 0 & 0 & 0 & 1_{(2)} & 0 & 0 & 1_{(1)} & 1_{(3)} \\
(e_1) & y & \boxed{1_{(3)}} & 0 & \boxed{1_{(2)}} & 0 & 0 & 0 & 0 & 0 \\
(e_1) & x^2 & 0 & 0 & 0 & 0 & 1_{(2)} & 1_{(1)} & 0 & 0 \\
(e_2) & 1 & \boxed{1_{(6)}} & 0 & \boxed{1_{(5)}} & 0 & 0 & 0 & 1_{(4)} & 0 \\
(e_2) & x^2 & 0 & 1_{(5)} & 0 & 0 & 0 & 1_{(4)} & 0 & 0
\end{array}
$$

The Factorisation Matrix

The factorised sum becomes a new expression and the original expressions are rewritten.

$$\begin{aligned} f &= 2x^2{}_{(7)} + 1_{(8)} \\ e_1 &= x^3{}_{(1)} + fy_{(9)} \\ e_2 &= x^3{}_{(4)} + f_{(10)} \end{aligned}$$

$$\begin{array}{cc|ccccc} & & 1 & f & fy & 2x^2 & x^3 \\ \hline (f) & 1 & 1_{(8)} & 0 & 0 & 1_{(7)} & 0 \\ (e_1) & 1 & 0 & 0 & 1_{(9)} & 0 & 1_{(1)} \\ (e_2) & 1 & 0 & 1_{(10)} & 0 & 0 & 1_{(4)} \end{array}$$

The Hosangadi et al. Algorithm

- Factorisations are chosen based on the number of floating point operations they save over the naive evaluation choice.
- For complex problems, the matrix can have hundreds of thousands of rows and columns.
- We represent the matrix as a bipartite graph so possible factorisations become bicliques within the graph.
- We've written an optimised branch and bound biclique search algorithm specific to our scoring function.
- Picking the best factorisation at each step means the algorithm is still *greedy*.
- Scalability is an issue for more complex forms.

Taking account of numeric relationships

- The Hosangadi et al. CSE pass is oblivious to numerical values.
- We want to be able to take advantage of numeric relationships. e.g.

$$e_0 = \frac{3}{5}x + \frac{5}{7}y \tag{3a}$$
$$e_1 = 1\frac{1}{5}x + 1\frac{3}{7}y \tag{3b}$$

- We decompose rationals into primes raised to positive and negative exponents:

$$e_0 = 3^1 5^{-1} x + 5^1 7^{-1} y \tag{4a}$$
$$e_1 = 3^1 2^1 5^{-1} x + 5^1 2^1 7^{-1} y \tag{4b}$$

- The extracted common sum c only needs to be computed once:

$$c = 3^1 5^{-1} x + 5^1 7^{-1} y \tag{5a}$$
$$e_0 = c \tag{5b}$$
$$e_1 = 2^1 c \tag{5c}$$

Exploiting exact knowledge of numerical values

- At every step of our analysis, we maintain our coefficients as *rational* numbers.
- We generate our Lagrange basis functions in the same way as implemented in FIAT, but solve the resulting linear system over the rationals.
- We must use symbolic integration rather than quadrature to evaluate the integral at code-generation time in order to preserve rational coefficients.
- We can now search for common subexpressions taking account of both distributivity of multiplication over addition and of numeric relationships between coefficients.

Some generated code...

```
void tabulate_tensor(double* const A, const double* const* w, const ufc::cell& c) const
{
  const double * const * x = c.coordinates;

  const double var_0 = -1.0000000000000000000000000*x[0][1];
  const double var_1 = x[2][1] + var_0;
  const double var_2 = -1.0000000000000000000000000*x[0][0];
  const double var_3 = x[1][0] + var_2;
  const double var_4 = var_0 + x[1][1];
  const double var_5 = var_2 + x[2][0];
  const double var_6 = var_1*var_3 + -1.0000000000000000000000000*var_4*var_5;
  const double var_7 = std::abs(var_6);
  const double var_8 = 0.0166666666666666664353702*var_7*w[0][0];
  const double var_9 = 0.0166666666666666664353702*var_7*w[0][1];
  const double var_10 = 0.0166666666666666664353702*var_7*w[0][2];
  const double var_11 = var_9 + var_10;
  A[5] = 0.0083333333333333332176851*var_7*w[0][0] + var_11;
  const double var_12 = var_9 + var_8;
  A[1] = 0.0083333333333333332176851*var_7*w[0][2] + var_12;
  A[3] = A[1];
  const double var_13 = var_10 + var_8;
  A[2] = 0.0083333333333333332176851*var_7*w[0][1] + var_13;
  A[6] = A[2];
  A[7] = A[5];
  A[4] = 0.0500000000000000027755576*var_7*w[0][1] + var_13;
  A[8] = 0.0500000000000000027755576*var_7*w[0][2] + var_12;
  A[0] = 0.0500000000000000027755576*var_7*w[0][0] + var_11;
}
```

Results (FLOP count from hardware performance counters)

We evaluated the operation count of quadrature, tensor and our generated local assembly implementations for various premultiplied mass matrices in 2D. e.g.

$a(u,v) = \int_\Omega f(x)g(x)h(x)(u(x)\cdot v(x))\,dx$

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p=1, q=1$	127	27	28	0.96	157	80	68	**1.18**	214	267	115	**1.86**	607	751	209	**2.90**
$p=1, q=2$	609	76	91	0.84	1123	193	163	**1.18**	1607	651	284	**2.29**	2682	1949	507	**3.84**
$p=1, q=3$	4935	126	161	0.78	7882	490	420	**1.17**	8057	1559	930	**1.68**	11851	3123	1211	**2.58**
$p=1, q=4$	17082	435	485	0.90	24847	1111	1060	**1.05**	25099	2542	2046	**1.24**	34503	4159	2794	**1.49**
$p=2, q=1$	151	49	55	0.89	583	315	219	**1.44**	1532	1970	926	**1.65**	2671	10637	2420	**1.10**
$p=2, q=2$	1111	117	131	0.89	2632	998	578	**1.73**	4255	5899	2346	**1.81**	-	-	-	-
$p=2, q=3$	7857	318	350	0.91	11779	1966	1425	**1.38**	16667	7860	4701	**1.67**	-	-	-	-
$p=2, q=4$	24811	853	978	0.87	34405	4306	3507	**1.23**	-	-	-	-	-	-	-	-
$p=3, q=1$	213	106	90	**1.18**	1607	1023	503	**2.03**	-	-	-	-	-	-	-	-
$p=3, q=2$	1607	223	217	**1.03**	4363	2743	1464	**1.87**	-	-	-	-	-	-	-	-
$p=3, q=3$	8057	756	853	0.89	16814	5684	4553	**1.25**	-	-	-	-	-	-	-	-
$p=3, q=4$	25099	1661	2015	0.82	45959	9856	9746	**1.01**	-	-	-	-	-	-	-	-

n_f is the number of premultiplying functions.
p is the degree of the premultiplying functions (e.g. f,g,h).
q is the degree of the basis functions (e.g. u,v).
FFC 0.9.10 with quadrature & tensor optimisations, GCC 4.6.1, with '-03' optimisation, Intel Core2 Duo.

Results (GNU C++ Compiler Timings, nano-seconds)

Timings don't include the cost of data movement or sparse matrix insertion.

$a(u, v) = \int_\Omega f(x)g(x)h(x)(u(x) \cdot v(x))\,dx$

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	69	19	19	**1.02**	86	46	26	**1.73**	109	129	40	**2.76**	387	734	112	**3.46**
$p = 1, q = 2$	627	67	69	0.97	1062	126	107	**1.18**	1500	551	193	**2.85**	2364	1851	361	**5.13**
$p = 1, q = 3$	2714	143	161	0.89	4221	379	267	**1.42**	4280	1149	729	**1.58**	6368	6273	935	**6.71**
$p = 1, q = 4$	8935	378	458	0.83	12858	1086	728	**1.49**	12957	2171	1712	**1.27**	17815	8719	6645	**1.31**
$p = 2, q = 1$	138	39	33	**1.16**	386	245	146	**1.68**	825	2029	651	**1.27**	1402	22184	1846	0.76
$p = 2, q = 2$	1076	83	82	**1.02**	2300	868	466	**1.86**	3445	12429	1896	**1.82**	-	-	-	-
$p = 2, q = 3$	4227	289	265	**1.09**	6344	1825	1147	**1.59**	8768	15883	10124	0.87	-	-	-	-
$p = 2, q = 4$	12930	709	763	0.93	17747	8843	8629	**1.02**	-	-	-	-	-	-	-	-
$p = 3, q = 1$	186	69	63	**1.10**	888	986	405	**2.19**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1518	148	143	**1.03**	3427	2811	1209	**2.33**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	4312	695	664	**1.05**	8668	11709	9890	0.88	-	-	-	-	-	-	-	-
$p = 3, q = 4$	13213	1336	1829	0.73	23659	22051	22617	0.98	-	-	-	-	-	-	-	-

n_f is the number of premultiplying functions.
p is the degree of the premultiplying functions (e.g. f,g,h).
q is the degree of the basis functions (e.g. u,v).
FFC 0.9.10 with quadrature & tensor optimisations, GCC 4.6.1, with '-03' optimisation, Intel Core2 Duo P8600 @ 2.4GHz.

Results (FLOP count from hardware performance counters)

The Intel C++ compiler is capable of optimising FFC generated local assembly implementations significantly more effectively than GCC.

	$n_f=1$				$n_f=2$				$n_f=3$				$n_f=4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p=1, q=1$	175	27	31	0.87	211	71	71	1.00	287	185	118	**1.57**	559	498	212	**2.35**
$p=1, q=2$	476	77	94	0.82	883	165	166	0.99	1271	425	286	**1.49**	2132	1066	510	**2.09**
$p=1, q=3$	2370	113	163	0.69	3849	340	417	0.82	3994	689	939	0.73	5947	1430	1213	**1.18**
$p=1, q=4$	7858	379	484	0.78	11538	661	1041	0.63	11879	1119	2060	0.54	16302	1667	2812	0.59
$p=2, q=1$	211	51	58	0.88	463	257	223	**1.15**	1157	1312	935	**1.24**	1951	7153	2507	0.78
$p=2, q=2$	667	125	134	0.93	1482	669	597	**1.12**	2422	3670	2434	1.00	-	-	-	-
$p=2, q=3$	3835	284	357	0.80	5659	1182	1452	0.81	8138	3901	4766	0.82	-	-	-	-
$p=2, q=4$	11536	639	978	0.65	15889	2067	3540	0.58	-	-	-	-	-	-	-	-
$p=3, q=1$	266	100	93	**1.08**	1157	761	524	**1.45**	-	-	-	-	-	-	-	-
$p=3, q=2$	1223	219	224	0.98	3123	1842	1511	**1.22**	-	-	-	-	-	-	-	-
$p=3, q=3$	3909	612	864	0.71	8101	2978	4613	0.65	-	-	-	-	-	-	-	-
$p=3, q=4$	11669	1223	2021	0.61	21325	5664	9819	0.58	-	-	-	-	-	-	-	-

n_f is the number of premultiplying functions.
p is the degree of the premultiplying functions (e.g. f,g,h).
q is the degree of the basis functions (e.g. u,v).
FFC 0.9.10 with tensor & quadrature optimisations, Intel C++ Compiler 11.1, with '-03' optimisation, Intel Core2 Duo.

Results (Intel C++ Compiler Timings, nano-seconds)

Timings don't include the cost of data movement or sparse matrix insertion.

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	110	19	23	0.81	126	35	26	**1.33**	172	79	40	**1.99**	323	233	72	**3.23**
$p = 1, q = 2$	591	49	55	0.88	1359	98	101	0.97	1902	250	151	**1.66**	3214	727	324	**2.25**
$p = 1, q = 3$	3536	139	170	0.82	5747	225	270	0.83	6025	443	599	0.74	8055	998	794	**1.26**
$p = 1, q = 4$	10543	342	479	0.71	15243	481	699	0.69	16045	754	1403	0.54	22867	1177	2888	0.41
$p = 2, q = 1$	208	38	38	1.00	469	161	125	**1.29**	1105	929	489	**1.90**	1927	16098	1341	**1.44**
$p = 2, q = 2$	868	85	91	0.93	1645	414	325	**1.27**	2706	7950	1402	**1.93**	-	-	-	-
$p = 2, q = 3$	5517	200	270	0.74	7900	742	873	0.85	11855	9052	9767	0.93	-	-	-	-
$p = 2, q = 4$	15350	462	740	0.62	21012	1480	7431	0.20	-	-	-	-	-	-	-	-
$p = 3, q = 1$	291	62	59	**1.05**	1140	587	327	**1.80**	-	-	-	-	-	-	-	-
$p = 3, q = 2$	1978	122	142	0.86	4867	1401	1035	**1.35**	-	-	-	-	-	-	-	-
$p = 3, q = 3$	5634	394	604	0.65	11241	7298	9593	0.76	-	-	-	-	-	-	-	-
$p = 3, q = 4$	15919	795	1567	0.51	28415	13605	20229	0.67	-	-	-	-	-	-	-	-

n_f is the number of premultiplying functions.
p is the degree of the premultiplying functions (e.g. f,g,h).
q is the degree of the basis functions (e.g. u,v).
FFC 0.9.10 with tensor & quadrature optimisations, Intel C++ Compiler 11.1, with '-03' optimisation, Intel Core2 Duo P8600 @ 2.4GHz.

Floating Point Inaccuracies

- To validate the correctness of our generated, we decided to compare against the FEniCS generated local assembly implementations.
- We noticed that for some forms, the results of the tensor contraction code deviated quite significantly from both the quadrature and our generated local assembly implementations.
- These deviations only occurred when the tensor contraction topological optimisations were enabled (co-linearity and Hamming distance analyses).

Comparison of error in local assembly implementations

	$n_f = 1$		$n_f = 2$		$n_f = 3$		$n_f = 4$	
	Excafé	Tensor	Excafé	Tensor	Excafé	Tensor	Excafé	Tensor
$p = 1, q = 1$	6.89e-17	1.40e-15	5.53e-17	9.90e-16	2.01e-17	8.53e-16	1.42e-17	4.43e-5
$p = 1, q = 2$	1.38e-16	9.68e-15	1.36e-16	1.60e-15	2.71e-17	2.07e-15	3.14e-17	7.11e-5
$p = 1, q = 3$	2.33e-16	3.86e-15	2.22e-16	6.09e-5	9.20e-17	5.30e-4	1.10e-16	2.62e-4
$p = 1, q = 4$	9.53e-16	2.00e-4	9.14e-16	4.60e-4	3.33e-16	7.20e-4	4.83e-16	3.70e-4
$p = 2, q = 1$	2.45e-16	8.34e-16	1.50e-16	1.41e-15	7.43e-17	1.30e-4	1.30e-16	2.03e-4
$p = 2, q = 2$	5.98e-16	1.82e-15	1.46e-16	7.04e-5	4.18e-16	3.08e-4	-	-
$p = 2, q = 3$	5.18e-16	1.65e-4	9.03e-16	6.37e-4	1.84e-15	1.41e-3	-	-
$p = 2, q = 4$	2.71e-15	1.01e-3	3.90e-15	1.71e-3	-	-	-	-
$p = 3, q = 1$	1.49e-16	1.60e-15	2.40e-16	6.24e-5	-	-	-	-
$p = 3, q = 2$	2.05e-16	2.99e-15	1.08e-15	6.31e-4	-	-	-	-
$p = 3, q = 3$	5.85e-16	3.26e-4	5.71e-15	1.24e-3	-	-	-	-
$p = 3, q = 4$	3.11e-15	1.11e-3	1.61e-14	2.48e-3	-	-	-	-

Basis function coefficients were chosen as random values between -1 and 1. Cell vertices were placed randomly on the unit circle.

Recap of FFC topological optimisations

Co-linearity:

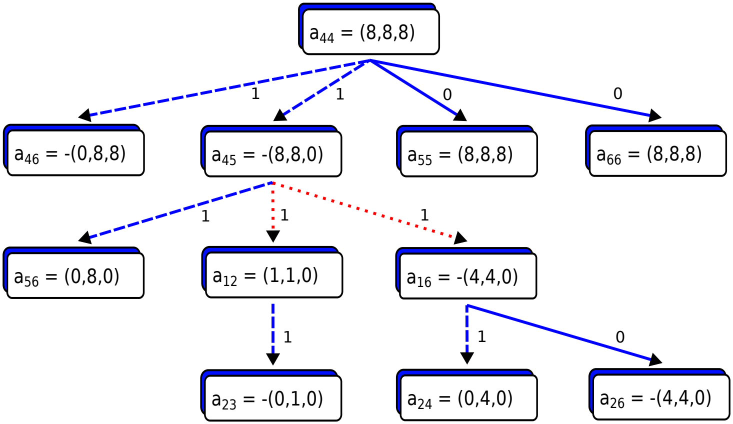
- The co-linearity optimisation computes local assembly matrix entries from each other using a scaling factor.

Hamming distance:

- The Hamming distance algorithm involves building a total graph whose nodes are the elements of the local assembly matrix.
- Each edge has a weight which represents the cost of computing one local assembly matrix entry from the other.
- The generated code corresponds to the computation described by the minimal spanning tree over this graph.

Complexity reducing relations

Dashed blue lines are Hamming distance optimisations.
Continuous blue lines are reuse of identical values.
Dashed red lines are colinearity optimisations.



What causes the error?

- Error appears to be caused by the Hamming distance optimisations rather than the colinearity ones.
- We note that for larger local assembly matrices, the minimal spanning tree will become larger.
- The larger the minimal spanning tree, the greater the accumulated numerical error as the inner products are updated.
- The accumulated error appears inherent to a system that cannot introduce new subexpressions.

Conclusion

- We have shown that for certain classes of variational forms, it's possible to reduce operation count over both tensor contraction and quadrature implementations by a factor of over 3.5 (GCC) or 2 (ICC).
- Actual performance improvements are dependent on architectural factors, form complexity and the amount of time spent performing local assembly.
- When we don't win, we still do better than the other lesser performing implementation.
- Tensor contraction topological optimisations can sometimes cause performance issues.
- For some forms, we have a significant reduction in operation count without any associated numerical precision issues.
- The Intel C++ Compiler can optimise both FFC-generated quadrature code and tensor contraction code significantly more effectively than GCC.