Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly

F. Luporini et al.

Presented by: Thilina Ratnayaka

December 7, 2018

PDEs

- Authors start by emphasizing the importance of PDEs, used in:
  - Computational Fluid Dynamics (e.g., INS)
  - Computational Electromagnetics
  - Structural Mechanics
- Numerical methods used:
  - FEM
  - FVM
- Authors claim that the time required to execute the computational kernels is a major issue, because:
  - large number of cells in the domain

FEM, Local Assembly

- Authors focus on FEM.
- They claim that local assembly is the most expensive part, consisting of 30% to 60% of total comuputation.
- During local assembly:
  - Contributions of a specific cell to the linear system is computed.
  - Involves evaluating problem-specific integrals to produce an element matrix and a vector.
- Authors try to optimize the local assembly phase, only talk about generating the local element matrix, excludes the vector.

Structure of a Local Assembly Kernel

```
Input: element matrix (2D array, initialized to 0), coordinates (array), coefficients (array, e.g. velocity)
Output: element matrix (2D array)
- Compute Jacobian from coordinates
- Define basis functions
- Compute element matrix in an affine loop nest
```

Fig. 1. Structure of a local assembly kernel.

FEM, Local Assembly, ctd.

- Authors focus on relatively low order finite element methods:
  - assembly kernels working set is usually small enough to fit in the L1 cache of traditional CPU.
- Authors claim that Low-order methods are employed in a wide variety of fields, including climate and ocean modeling, computational fluid dynamics, and structural mechanics.
- Also authors exclude the GPUs from the study as well.

Assembly Kernel, ctd.

- Authors claim that in low order FEM loop nests,
  - Individual loops are rather small (between 3 and 30).
  - Local element matrix is evaluated at the innermost loop.
- Authors claim that transformations for cache locality (e.g., blocking) are not helpful.
- Instead, authors focus on
  - Optimization of floating-point operations.
  - Register locality.
  - Instruction-level parallelism (vectorization).

Methodology

- Authors have automated a set of generic and model-driven code transformations in COFFEE:
  - A compiler for optimizing local assembly kernels.
  - Uses kernel from [Firedrake 2014], a system for solving PDEs using FEM, as input.
- In evaluating code transformations they vary:
  - Polynomial order.
  - Geometry of elements.

Local assembly code generated by Firedrake

- Given a finite element input problem expressed by domain-specific Unified Form Language (UFL):
  - Firedrake uses FEniCS form compiler (FFC) to generate an abstract syntax tree (AST) of a assembly kernel.
  - then compiles the kernel using an available vendor compiler.
- The main contribution of the paper is to enhance this execution model by adding an optimization stage prior to the generation of C code.

Example: Helmholtz problem

LISTING 1: Local assembly source code generated by Firedrake for a Helmholtz problem on a 2D triangular mesh using Lagrange $p = 1$ elements.

```
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]);
}
```

Example: Helmholtz problem, ctd.

Table I. Type and Variable Names Used in the Various Listings to Identify Local Assembly Objects

Object Name	Type	Variable Name(s)
Determinant of the Jacobian matrix	double	det
Inverse of the Jacobian matrix	double	K1, K2, ...
Coordinates	double**	coords
Fields (coefficients)	double**	w
Numerical integration weights	double[]	W
Basis functions (and derivatives)	double[][]	X, Y, X1, ...
Element matrix	double[][]	A

Padding and Data Alignment

- Authors claim that the absence of stencils makes element matrix computation easily auto vectorizable.
- Auto vectorization is not efficient if:
  - data are not aligned to cache-line boundaries
  - the length of the innermost loop is not a multiple of the vector length VL
- They enforce data alignment in two steps:
  - First, all arrays are allocated to addresses that are multiples of VL.
  - Then, 2D arrays are padded by rounding the number of columns to the nearest multiple of VL.

Padding and Data Alignment, ctd.

- For example on AVX processor with 256 bit long vector and 64 bit doubles, 3x3 basis function array will have size 3x4.
- The compiler is explicitly notified about this using pragmas.
  - For intel compilers, #pragma vector aligned is added.
- This causes the compiler to issue aligned data loads and stores.

Generalized Loop-Invariant Code Motion

- In the Helmholtz equation, computation of local element matrix only depends on two iteration variables.
- Authros claim that Vendor compilers only identify subexpressions that are invariant with respect to the innermost loop.

Generalized Loop-Invariant Code Motion, ctd.

- Authors work around these limitations with source-level loop-invariant code motion.
- They pre-compute all values that an invariant subexpression assumes along its fastest varying dimension.
- This is implemented by introducing a temporary array per invariant subexpression and by adding a new loop to the nest.

Generalized Loop-Invariant Code Motion, ctd.

LISTING 3: Local assembly source code for the Helmholtz problem in Listing 1 after application of padding, data alignment, and loop-invariant code motion, for an AVX architecture. In this example, subexpressions invariant to `j` are identical to those invariant to `k`, so they can be precomputed once in the *r* loop.

```
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];
 }
}
```

Expression Splitting

- In complex kernels, and on certain architectures, achieving effective register allocation can be challenging.
- If the number of variables independent of the innermost-loop dimension is close to or greater than the number of available CPU registers, then poor register reuse is likely.
- One potential solution to this problem consists of suitably splitting the computation.

Expression Splitting, ctd.

LISTING 5: Local assembly source code generated by Firedrake for the Helmholtz problem in which *split* has been applied on top of the optimizations shown in Listing 3. In this example, the split factor is 2.

```
void helmholtz(double A[3][4], double **coords) {
 // Same code as in Listing~3 up to the j loop
 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])*det*W[i];
 for (int j = 0; j<3; j++)
  #pragma vector aligned
 for (int k = 0; k<4; k++)
   A[j][k] += LI_1[k]*LI_1[j]*det*W[i];
}
```

Overview of COFFEE

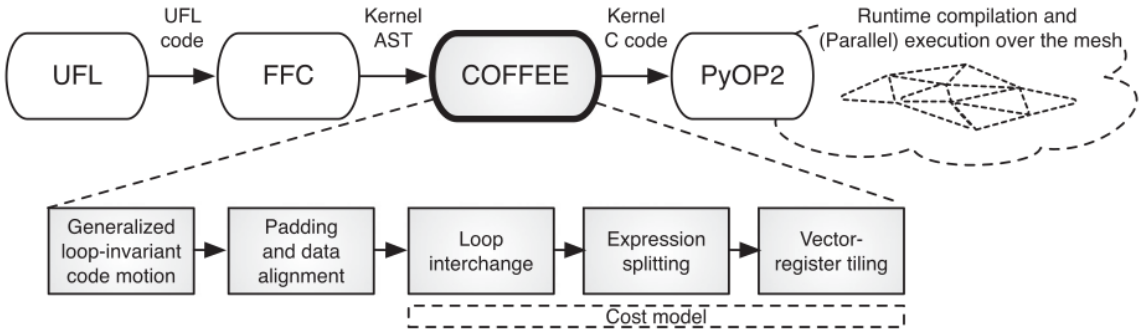

Fig. 4. High-level view of Firedrake. COFFEE is at the core, receiving ASTs from a modified version of the FFC and producing optimized C code kernels.

How COFFEE works

- COFFEE applies an ordered sequence of optimization steps to ASTs recieved from FFC.
- Loop invariant code motion and padding is always performed in that order.
- Loop interchange, expression splitting and vector register tiling is introduced based on a cost model.

Loop interchanges and unroll

- Loop interchanges are done such that the number of invariant loads are smallest.
- Manual full unroll will exceed the instruction cache and inhibit compiler auto-vectorization and hence not effective.

Cost Model

```
1  Input: n_outer_arrays, n_inner_arrays, n_consts, n_regs
2  Output: uaj_factor, split_factor
3  n_outer_regs = n_regs / 2
4  split_factor = 0
5  // Compute spltting factor
6  while n_outer_arrays > n_outer_regs
7    n_outer_arrays = n_outer_arrays / 2
8    split_factor = split_factor + 1
9  // Compute unroll-and-jam factor for op-vect
10 n_regs_avail = n_regs - (n_outer_arrays + n_consts)
11 uaj_factor = n_reg_avail / n_inner_arrays
12 // Estimate the benefit of permuting loops
13 permute = n_outer_arrays > n_inner_arrays
14 return <permute, split_factor, uaj_factor>
```

- **n_regs**: number of registers available.
- **n_consts**: variables independent of j and k.
- **n_inner_arrays**: k-dependent variables.
- **n_outer_arrays**: j-dependent variables.

Experimental Setup

- Experiments were run on a single core of two Intel architectures: a Sandy Bridge (I7-2600 CPU, running at 3.4GHz, 32KB L1 cache, and 256KB L2 cache) and a Xeon Phi (5110P, running at 1.05GHz in native mode, 32KB L1 cache, and 512KB L2 cache).
- These two architectures have been choosen because of the differences in the number of logical registers and SIMD lanes (16 256-bit registers in the Sandy Bridge and 32 512-bit registers in the Xeon Phi), which can impact the optimization strategy.
- The icc 13.1 compiler was used. Authors selected the best optimization levels for each platform; -xAVX for autovectorization and -O2 on the Sandy Bridge, and -O3 on the Xeon Phi.

Results

- Three problems were studied varying both the shape of mesh elements and the polynomial order p of the method.
- Results are only shown for **ijk** loop order. Other interchanges did not make any significant improvement according to the authors.

Loop Invariant Code Motion (LICM)

Table II. Performance Improvement Due to Generalized Loop-Invariant Code Motion over the Original Nonoptimized Code

		Sandy Bridge				Xeon Phi			
problem	shape	p1	p2	p3	p4	p1	p2	p3	p4
Helmholtz	triangle	1.05	1.46	1.68	1.67	1.49	1.06	1.05	1.17
Helmholtz	tetrahedron	1.36	2.10	2.64	2.27	1.28	1.29	2.05	1.73
Helmholtz	prism	2.16	2.28	2.45	2.06	1.04	2.26	1.93	1.64
Diffusion	triangle	1.09	1.68	1.97	1.64	1.07	1.06	1.18	1.16
Diffusion	tetrahedron	1.30	2.20	3.12	2.60	1.00	1.38	2.02	1.74
Diffusion	prism	2.15	1.82	2.71	2.32	1.11	2.16	1.85	2.83
Burgers	triangle	1.53	1.81	2.68	2.46	1.21	1.42	2.34	2.97
Burgers	tetrahedron	1.61	2.24	1.69	1.59	1.01	2.55	0.98	1.21
Burgers	prism	2.11	2.20	1.66	1.32	1.39	1.56	1.18	1.04

- Gain tend to increase with computational cost of the kernels.

LICM + Padding and Data Alignment - LICM-AP

Table III. Performance Improvement Due to Generalized Loop-Invariant Code Motion, Data Alignment, and Padding over the Original Nonoptimized Code

		Sandy Bridge				Xeon Phi			
problem	shape	p1	p2	p3	p4	p1	p2	p3	p4
Helmholtz	triangle	1.32	1.88	2.87	4.13	1.50	2.41	1.30	1.96
Helmholtz	tetrahedron	1.35	3.32	2.66	3.27	1.41	1.50	2.79	2.81
Helmholtz	prism	2.63	2.74	2.43	2.75	2.38	2.47	2.15	1.71
Diffusion	triangle	1.38	1.99	3.07	4.28	1.08	1.88	1.20	1.97
Diffusion	tetrahedron	1.41	3.70	3.18	3.82	1.05	1.51	2.76	3.00
Diffusion	prism	2.55	3.13	2.73	2.69	2.41	2.52	2.05	2.48
Burgers	triangle	1.56	2.28	2.61	2.77	2.84	2.26	3.96	4.27
Burgers	tetrahedron	1.61	2.10	1.60	1.78	1.48	3.83	1.55	1.29
Burgers	prism	2.19	2.32	1.64	1.42	2.18	2.82	1.24	1.25

LICM-AP + Expression splitting

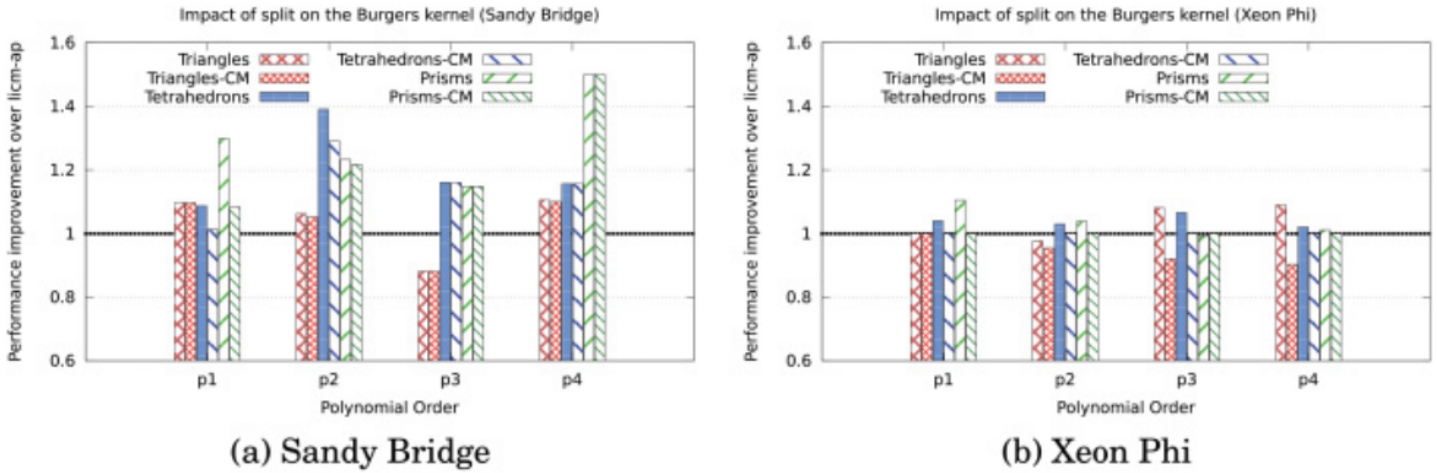

Fig. 8. Performance improvement over *licm-ap* obtained by *split* in the Burgers kernel. Bars suffixed with "CM" indicate that the cost model was used to transform the kernel.