A few notes

immediate

1 Prototyping a language and a compiler for unstructured mesh computations

This section tries to delineate the research path done so far and the potential future work. In no ways these things will be useful for the next (if any) COFFEE paper, but hopefully they will clarify my (and your) mind about the status of the research. As of October 2013, I have worked on two projects

- Sparse tiling for irregular computations: ST
- A compiler for optimizing the sequential code of finite element integration (local assembly): COFFEE

Despite the original motivations for the two projects arise from different areas - finite element for COFFEE, finite volume/molecular dynamics and, more in general, irregular computations for ST - I think much ground is in common. For example, the programming (at least at a certain level of abstraction) and execution models of the motivating applications are identical: OP2 establishes a way of defining *loop chains*, in which, as you all know, the computation is structured as a sequence of synchronized calls to kernels applied all over the mesh. Given that, ST and COFFEE can be abstracted into a single framework. Such a framework would formalize the way we optimize unstructured mesh applications.

What do I mean by framework? This is all about being aware of the domain in which we operate, in order to be able to introduce powerful optimizations.

- Unstructured mesh computations are, in a certain sense, special: they can be *always* represented as a sequence of loop nests, each loop nest being characterized by the presence of a non-affine outer loop (over mesh elements). We all know that compilers tend to fail at optimizing non-affine loops, for obvious reasons. We want to be in a position that allows us to optimize individual, non-affine loops.
- Such a sequence of loop nests is known, at compile/code-generation time, and described by a loop chain abstraction, which also tells how the non-affine accesses are distributed (by a sequence of *opparloop* calls in OP2). We want to exploit the knowledge of the loop chain, e.g. for inspector/executor strategies.

- In specific domains, kernels have a particular structure: for example, an affine (perfect or non-perfect) loop nest, in which mathematical expressions have certain properties, loops are fully parallel, etc. We want to exploit domain-knowledge.

By themselves, these observations do not introduce anything new: however, what is important here is the formalization of a framework that allows exploiting these three different potential sources of optimization in a systematic way.

It is worth highlighting the differences with respect to the standard OP2-like programming model:

- We will have two domain-specific languages: one is obviously an OP2-like language to write loop chains; the other, let's call it X for the moment, will be used to drive the optimization process. The optimization process may involve transforming the kernel code, as well as specializing the wrapper code.
- X is used to specify multi-level optimizations:
  1. Across the loop chain (e.g. ST, kernel fusion, kernel fission)
  2. Within a single loop of the loop chain (e.g. Inter-kernel vectorization;)
  3. Within a kernel (e.g. COFFEE's optimizations, i.e. domain-aware optimizations [1])

  And all of these optimizations are composable. This is summarized in Figure **??**, along with some examples of X constructs expressed as pragma notation.
- The output of our framework would be the input to the plan function, which would be responsible for scheduling the parallelism exposed in the input (e.g. standard execution of parallel loops with a synchronization between the execution of two loops, with kernels executed sequentially; parallel (threaded) execution of a kernel, if X exposed intra-kernel parallelism; lazyness; various techniques for communication-computation overlap; etc.).

2 Optimizing finite element integration through expression rewriting and code specialization

This is basically a **generalization** of the COFFEE paper. We claim an optimizer for local assembly routines needs to go through two neatly separated steps: expression rewriting and code specialization.

- **Expression rewriting.** *Intuitively*, at this level we do transformations that minimize the amount of redundant computation. This is an interesting problem that involves several steps (here in random order):

[1] note that domain-aware is different than domain-specific.

1. factorization of common terms (a*c + a*b → a*(b+c))
2. loop-invariant code motion (note that factorization at point 1 can expose more invariant terms)
3. expansion of terms (note that this can expose more factorization opportunities - e.g. (b+c)*a + b*d + c*e→ b*(a + c) + c*(a + e))
4. eliminate arithmetic operations involving zero-valued terms (this requires re-writing loop bounds, splitting expressions, etc)

Of these, only point 2 was presented in the COFFEE paper. The idea here is to formalize a set of rewriting rules (similar in spirit to SPIRAL ones? maybe not...) that deterministically bring a given mathematical expression from its original form to a new form in which we have to do less floating point operations.

- **Code specialization.** In this phase, we specialize and optimize code for the underlying architecture. We have several possibilities. Some of them were in the COFFEE paper already.

  1. Padding, data alignment, trip count adjustment. This is for improving simdization.
  2. Loop fission based on expression splitting (i.e. on commutativity of some operations)
  3. Outer-product vectorization

  The outer-product vectorization is just an example of something that can be generalized as: we know the memory access pattern of the expression, can we write specialized code for it? In general, we may think to ask a compiler "Hey, it's really important to optimize the execution of this expression in this loop nest, can you check whether you know how to generate specialized code for this notably important memory access pattern?". So the outer-product vectorization becomes a possible optimized implementation of a given expression in a certain iteration space. Another possibility is to transform the computation, for instance, in a sequence to BLAS calls. So, we have more opportunities here. On top of these things, we have to consider that many more code variants should be tried because:

  – we may want to try a different loop interchange (partly neglected in the COFFEE paper)
  – we may want to manually unroll-and-jam loops (neglected in the COFFEE paper)
  – we may want to try different ways of re-writing the expression! Especially the elimination of arithmetic operations involving zero-valued terms poses several questions since it leads to break the original iteration space into multiple loop nests, in which padding/data alignment/trip count adjustment would not be safe anymore. For example, compare Figures 1 and 2. In the transformed code in Figure 2, the first two `jk` loop nests in which A is evaluated are not decorated with `pragma alignment/simd` because adjusting loop bounds would not be safe and accesses would not be aligned to the vector length. This is a pity because alignment/padding, as shown in the COFFEE paper, have a great impact. To make use of them, the only possibility is to store partial evaluations from different loops in temporary arrays, and then store them back into A at the end of the

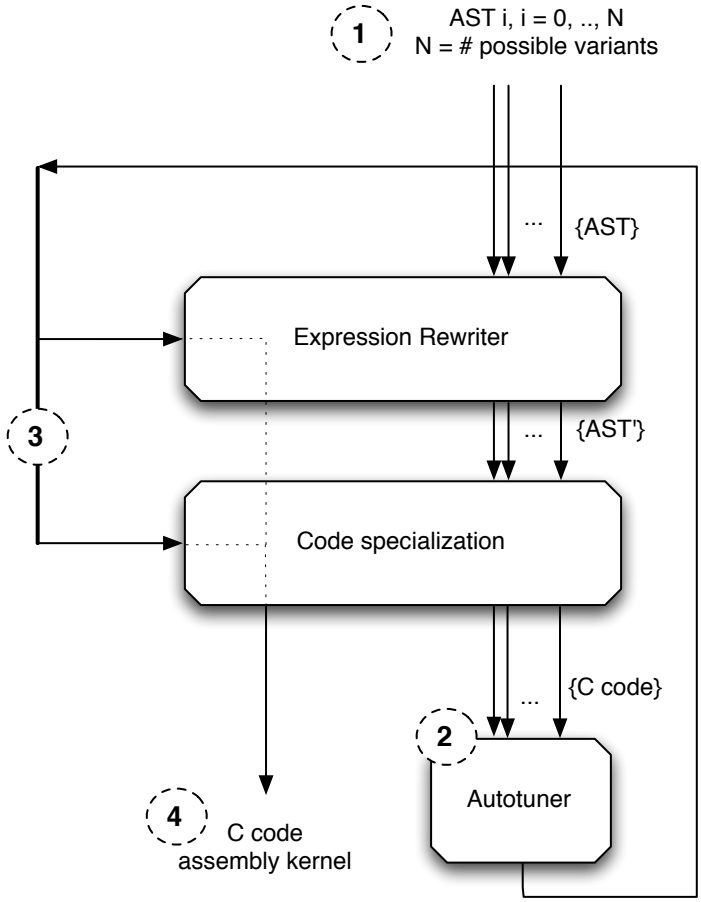

Figure 1: Overview of COFFEE. Numbers 1-2-3-4 indicate the flow.

computation (see Figure 3). So what is the overhead of this approach? What is it best to do? Should we lose the benefit of padding/alignment, or should we use temporary arrays as in Figure 3, or should we just forget about the elimination of zeros (*in a given problem*)?

Therefore, a possible thing to do for specializing code in a certain problem is

– Autotuning, based on domain-knowledge to keep the exploration space small enough

which would replace the cost model introduced in the COFFEE paper (here, we have more code variants). A scheme of the new COFFEE is in Figure 2

Figure 2: Original code

```
for (int ip  = 0; ip < 8; ++ip)
{
  for (int j  = 0; j < 18; ++j)
  {
    for (int k  = 0; k < 20; ++k)
    {
      A[j][k] += (((((K[2] * FE0_C2_D100[ip][k]) + (K[5] * FE0_C2_D010[ip][k]) + (
          K[8] * FE0_C2_D001[ip][k])) * ((K[2] * FE0_C2_D100[ip][j]) + (K[5] *
          FE0_C2_D010[ip][j]) + (K[8] * FE0_C2_D001[ip][j]))) + (((K[1] *
          FE0_C2_D100[ip][k]) + (K[4] * FE0_C2_D010[ip][k]) + (K[7] * FE0_C2_D001[
          ip][k])) * ((K[1] * FE0_C2_D100[ip][j]) + (K[4] * FE0_C2_D010[ip][j]) +
          (K[7] * FE0_C2_D001[ip][j]))) + (((K[0] * FE0_C2_D100[ip][k]) + (K[3] *
          FE0_C2_D010[ip][k]) + (K[6] * FE0_C2_D001[ip][k])) * ((K[0] *
          FE0_C2_D100[ip][j]) + (K[3] * FE0_C2_D010[ip][j]) + (K[6] * FE0_C2_D001[
          ip][j]))) + (((K[2] * FE0_C1_D100[ip][k]) + (K[5] * FE0_C1_D010[ip][k])
          + (K[8] * FE0_C1_D001[ip][k])) * ((K[2] * FE0_C1_D100[ip][j]) + (K[5] *
          FE0_C1_D010[ip][j]) + (K[8] * FE0_C1_D001[ip][j]))) + (((K[1] *
          FE0_C1_D100[ip][k]) + (K[4] * FE0_C1_D010[ip][k]) + (K[7] * FE0_C1_D001[
          ip][k])) * ((K[1] * FE0_C1_D100[ip][j]) + (K[4] * FE0_C1_D010[ip][j]) +
          (K[7] * FE0_C1_D001[ip][j]))) + (((K[0] * FE0_C1_D100[ip][k]) + (K[3] *
          FE0_C1_D010[ip][k]) + (K[6] * FE0_C1_D001[ip][k])) * ((K[0] *
          FE0_C1_D100[ip][j]) + (K[3] * FE0_C1_D010[ip][j]) + (K[6] * FE0_C1_D001[
          ip][j]))) + (((K[2] * FE0_C0_D100[ip][k]) + (K[5] * FE0_C0_D010[ip][k])
          + (K[8] * FE0_C0_D001[ip][k])) * ((K[2] * FE0_C0_D100[ip][j]) + (K[5] *
          FE0_C0_D010[ip][j]) + (K[8] * FE0_C0_D001[ip][j]))) + (((K[1] *
          FE0_C0_D100[ip][k]) + (K[4] * FE0_C0_D010[ip][k]) + (K[7] * FE0_C0_D001[
          ip][k])) * ((K[1] * FE0_C0_D100[ip][j]) + (K[4] * FE0_C0_D010[ip][j]) +
          (K[7] * FE0_C0_D001[ip][j]))) + (((K[0] * FE0_C0_D100[ip][k]) + (K[3] *
          FE0_C0_D010[ip][k]) + (K[6] * FE0_C0_D001[ip][k])) * ((K[0] *
          FE0_C0_D100[ip][j]) + (K[3] * FE0_C0_D010[ip][j]) + (K[6] * FE0_C0_D001[
          ip][j])))) * det * W8[ip]);
    }
  }
}
```

Figure 3: Code after expression re-writing. Operations involving zero-valued entries in the arrays are removed by suitably adjusting the iteration space.

```
for (int ip  = 0; ip < 8; ++ip)
{
  double LI_IPJ_1_0[20] __attribute__((aligned(32))), LI_IPJ_1_1[20] __attribute__
      ((aligned(32))), LI_IPJ_1_2[20] __attribute__((aligned(32))), LI_IPJ_1_3[20]
       __attribute__((aligned(32))), LI_IPJ_1_4[20] __attribute__((aligned(32))),
      LI_IPJ_1_5[20] __attribute__((aligned(32))), LI_IPJ_1_6[20] __attribute__((
      aligned(32))), LI_IPJ_1_7[20] __attribute__((aligned(32))), LI_IPJ_1_8[20]
      __attribute__((aligned(32))) ;
  #pragma simd
  #pragma vector aligned
  for (int j  = 0; j < 20; ++j)
  {
    LI_IPJ_1_0[j] = ((K[1] * FE0_C2_D100[ip][j]) + (K[4] * FE0_C2_D010[ip][j]) + (
        K[7] * FE0_C2_D001[ip][j]));
    LI_IPJ_1_1[j] = ((K[0] * FE0_C0_D100[ip][j]) + (K[3] * FE0_C0_D010[ip][j]) + (
        K[6] * FE0_C0_D001[ip][j]));
    ...
  }

  double LI_IPK_1_0[20] __attribute__((aligned(32))), LI_IPK_1_1[20] __attribute__
      ((aligned(32))), LI_IPK_1_2[20] __attribute__((aligned(32))), LI_IPK_1_3[20]
       __attribute__((aligned(32))), LI_IPK_1_4[20] __attribute__((aligned(32))),
      LI_IPK_1_5[20] __attribute__((aligned(32))), LI_IPK_1_6[20] __attribute__((
      aligned(32))), LI_IPK_1_7[20] __attribute__((aligned(32))), LI_IPK_1_8[20]
      __attribute__((aligned(32)));
  double const0  = det * W8[ip];
  #pragma simd
  #pragma vector aligned
  for (int k  = 0; k < 20; ++k)
  {
    LI_IPK_1_0[k] = (((K[2] * FE0_C0_D100[ip][k]) + (K[5] * FE0_C0_D010[ip][k]) +
        (K[8] * FE0_C0_D001[ip][k])) * const0);
    LI_IPK_1_1[k] = (((K[1] * FE0_C2_D100[ip][k]) + (K[4] * FE0_C2_D010[ip][k]) +
        (K[7] * FE0_C2_D001[ip][k])) * const0);
    ...
  }

  // Note the original j-k loop nest is now broken into multiple nests to avoid
      operations involving zero terms. It was done in a way to preserve
      vectorizability, i.e. eliminating only contiguous zero-columns, which
      preserves stride-1 memory accesses.

  for (int j  = 6; j < 12; ++j)
  {
    for (int k  = 6; k < 12; ++k)
    {
      A[j][k] += ((LI_IPK_1_6[k] * LI_IPJ_1_4[j]));
      A[j][k] += ((LI_IPK_1_5[k] * LI_IPJ_1_5[j]));
      A[j][k] += ((LI_IPK_1_2[k] * LI_IPJ_1_6[j]));
    }
  }
  for (int j  = 0; j < 6; ++j)
  {
    for (int k  = 0; k < 6; ++k)
    {
      A[j][k] += ((LI_IPK_1_3[k] * LI_IPJ_1_1[j]));
      A[j][k] += ((LI_IPK_1_0[k] * LI_IPJ_1_2[j]));
      A[j][k] += ((LI_IPK_1_4[k] * LI_IPJ_1_8[j]));
    }
  }
  for (int j  = 12; j < 18; ++j)
  {
    #pragma simd
    #pragma vector aligned
    for (int k  = 12; k < 20; ++k)
    {
      A[j][k] += ((LI_IPK_1_1[k] * LI_IPJ_1_0[j]));
      A[j][k] += ((LI_IPK_1_7[k] * LI_IPJ_1_3[j]));
      A[j][k] += ((LI_IPK_1_8[k] * LI_IPJ_1_7[j]));
    }
  }
}
```

Figure 4: Code after expression re-writing plus using temporaries to ensure aligned accesses.

```
for (int ip  = 0; ip < 8; ++ip)
{
  ... // Same code as in Figure 2

  // Note the followings are padded from 6 to 8 for data alignment and for
       avoiding peeling loops
  double TMP_1[6][8] __attribute__((aligned(32)));
  double TMP_2[6][8] __attribute__((aligned(32)));

  for (int j  = 6; j < 12; ++j)
  {
    #pragma simd
    #pragma vector aligned
    for (int k  = 6; k < 14; ++k) // Note increased from 12 to 14
    {
      TMP_1[j][k] += ((LI_IPK_1_6[k] * LI_IPJ_1_4[j]));
      TMP_1[j][k] += ((LI_IPK_1_5[k] * LI_IPJ_1_5[j]));
      TMP_1[j][k] += ((LI_IPK_1_2[k] * LI_IPJ_1_6[j]));
    }
  }
  for (int j  = 0; j < 6; ++j)
  {
    #pragma simd
    #pragma vector aligned
    for (int k  = 0; k < 8; ++k) // Note increased from 6 to 8
    {
      TMP_2[j][k] += ((LI_IPK_1_3[k] * LI_IPJ_1_1[j]));
      TMP_2[j][k] += ((LI_IPK_1_0[k] * LI_IPJ_1_2[j]));
      TMP_2[j][k] += ((LI_IPK_1_4[k] * LI_IPJ_1_8[j]));
    }
  }
  for (int j  = 12; j < 18; ++j)
  {
    #pragma simd
    #pragma vector aligned
    for (int k  = 12; k < 20; ++k)
    {
      A[j][k] += ((LI_IPK_1_1[k] * LI_IPJ_1_0[j]));
      A[j][k] += ((LI_IPK_1_7[k] * LI_IPJ_1_3[j]));
      A[j][k] += ((LI_IPK_1_8[k] * LI_IPJ_1_7[j]));
    }
  }
}

// Need to copy from temporaries to A; note this is outside of the i loop
for (int j  = 0; j < 6; ++j)
{
  for (int k  = 0; k < 6; ++k)
  {
    A[j+6][k+6] = TMP_1[j][k];
    A[j][k] += TMP_2[j][k];
  }
}
```