arXiv:1705.03667v1 [cs.MS] 10 May 2017

TSFC: A STRUCTURE-PRESERVING FORM COMPILER*

MIKLÓS HOMOLYA†, LAWRENCE MITCHELL‡, FABIO LUPORINI§, AND DAVID A. HAM¶

Abstract. A *form compiler* takes a high-level description of the weak form of partial differential equations and produces low-level code that carries out the finite element assembly. In this paper we present the *Two-Stage Form Compiler* (TSFC), a new form compiler with the main motivation to maintain the structure of the input expression as long as possible. This facilitates the application of optimizations at the highest possible level of abstraction. TSFC features a novel, structure-preserving method for separating the contributions of a form to the subblocks of the local tensor in discontinuous Galerkin problems. This enables us to preserve the tensor structure of expressions longer through the compilation process than other form compilers. This is also achieved in part by a two-stage approach that cleanly separates the lowering of finite element constructs to tensor algebra in the first stage, from the scheduling of those tensor operations in the second stage. TSFC also efficiently traverses complicated expressions, and experimental evaluation demonstrates good compile-time performance even for highly complex forms.

Key words. code generation, finite element method, form compiler, tensor algebra, weak form

AMS subject classifications. 68N20, 65M60, 65N30

1. Introduction.

The development of state-of-the-art finite element simulation software has become an especially demanding endeavour. Advanced finite element techniques are particularly complicated, the efficient utilization of modern computers requires onerous effort, and the application domain itself is often a source of remarkable complexity. Automatic code generation techniques have been employed by various software projects to decouple and manage these complexities. Examples include Analysa [5], GetDP [10], Sundance [29], FreeFem++ [14], the FEniCS Project [27, 2], and Firedrake [35]. The finite element method is particularly suited to code generation approaches since the entire mathematical specification of a finite element problem can be expressed at a high level of abstraction. The specification of the weak form of the equations, along with the discrete function space from which each variable is to be drawn, is sufficient to characterise the problem completely. The *Unified Form Language* (UFL) [3] is a domain specific language for the finite element method, and provides the input language for FEniCS and Firedrake. UFL captures the weak form and discretised function spaces, thereby providing the information that the code generation system requires.

The solution of a finite element problem requires three distinct operations. First, each integral in the problem (that is, each multilinear form) must be evaluated on each cell or facet of the mesh. Second, the facet and cell components are assembled into global matrices and/or vectors as appropriate. Finally, these global objects are

*Submitted to SIAM Journal on Scientific Computing on May 15, 2017.

Funding: This work was supported by The Grantham Institute; the Engineering and Physical Sciences Research Council [grant number EP/M011054/1]; the Department of Computing, Imperial College London; and the Natural Environment Research Council [grant number NE/K008951/1].

†The Grantham Institute and Department of Computing, Imperial College London, London, SW7 2AZ, UK (m.homolya14@imperial.ac.uk)

‡Department of Computing and Department of Mathematics, Imperial College London, London, SW7 2AZ, UK (lawrence.mitchell@imperial.ac.uk)

§Now at Department of Earth Science and Engineering, Imperial College London, London, SW7 2AZ, UK (f.luporini12@imperial.ac.uk)

¶Department of Mathematics, Imperial College London, London, SW7 2AZ, UK (david.ham@imperial.ac.uk)

used by a linear solver to compute the answer to the system. Time varying and nonlinear PDEs are solved by composing these operations under the control of a time integration scheme and nonlinear solver respectively.

A *form compiler* takes a multilinear form expressed in UFL and produces low-level code that assembles the form on a single cell or facet of the mesh. The current production form compiler in FEniCS is the *FEniCS Form Compiler* (FFC) [19, 28] while the *SyFi Form Compiler* (SFC) [4] was an earlier alternative. The *UFL Analyser and Compiler System* (UFLACS) has been developed as a replacement for the previous FFC implementations, and has recently become default. In this paper we present the *Two-Stage Form Compiler* (TSFC), a new form compiler in use in the Firedrake project. The main motivation in TSFC is to maintain the structure of the input expression as long as possible to facilitate the application of optimizations at the highest possible level of abstraction.

The contributions of TSFC, and therefore of this paper, are a number of algorithmic advances over the state of the art form compilers currently available. By efficiently traversing the input UFL expressions, the compile time of TSFC scales much better than legacy FFC as form complexity increases and in the cases where the transformation from the reference cell to the mesh is non-affine. TSFC preserves the tensor structure of expressions for longer through the compilation process than either FFC or UFLACS. In particular, this is achieved by a new approach to isolating the components of a weak form which must be inserted into different rows and columns of the output tensor. Finally, TSFC cleanly separates the lowering of finite element constructs to tensor algebra on the one hand, from the scheduling of those tensor operations on the other. This facilitates the optimisation of this schedule either in situ or by a downstream tool.

The rest of this work is arranged as follows. In the remainder of this section we review some concepts necessary for understanding form compilers, and then give an overview of other current form compilers. In section 2 we emphasize the importance of preserving structure, and propose a novel, structure-preserving method for an issue that arises in discontinuous Galerkin problems. We compare this with the approach adopted by UFLACS, and consider their compatibility with our existing optimization technology. Then we discuss the first compilation stage in section 3, briefly describing the preprocessing in UFL, then explaining the translation of the preprocessed UFL form to the intermediate tensor algebra language. Section 4 then gives a detailed description of this intermediate representation along with its translation to low-level C code. Section 5 discusses optimizations employed at various stages of the form compilation pipeline. Experimental evaluation is presented in section 6, and section 7 finally concludes the paper.

1.1. The role of a form compiler. Consider the stationary heat equation $-\nabla \cdot (\kappa \nabla u) = f$ with κ thermal conductivity and f heat source on a domain Ω with $u = 0$ on the boundary. Its standard weak form with test function $v \in \hat{V}$ can be written as

$$a(v, u) = L(v) \tag{1.1}$$

where a is a bilinear and L is a linear form defined as

$$a(v, u; x, \kappa) = \int_\Omega \kappa \nabla u \cdot \nabla v \, \mathrm{d}x \quad \text{and} \tag{1.2}$$

$$L(v; x, f) = \int_\Omega f v \, \mathrm{d}x. \tag{1.3}$$

Where κ and f are prescribed spatial functions that parametrize the forms a and L, and x represents the coordinate field that characterizes the domain Ω. Following the UFL terminology, we refer to v and u as *arguments*, and to x, κ, and f as *coefficients*. A form must be linear in its arguments, but not necessarily in its coefficients.

Let V and $\hat{V}$ be suitable finite-dimensional spaces of functions on Ω and let their bases be $\{\phi_i\}_{i=1}^{s}$ and $\{\hat{\phi}_i\}_{i=1}^{\hat{s}}$. Seeking a solution $u(x) = \sum_{i=1}^{s} \mathbf{u}_i \phi_i(x)$ in the solution space V reduces (1.1) to a linear system $\mathbf{A} \cdot \mathbf{u} = \mathbf{b}$ with

$$\mathbf{A}_{ij} = a(\hat{\phi}_i, \phi_j; x, \kappa) \quad \text{and} \tag{1.4}$$

$$\mathbf{b}_i = L(\hat{\phi}_i; x, f). \tag{1.5}$$

$\mathbf{A}$ is the bilinear form a assembled as a matrix, and $\mathbf{b}$ is the linear form L assembled as a vector.

UFL is a domain-specific language for the description of the weak form and discretized function spaces of finite element problems. UFL is embedded in Python. Listing 1 defines a and L in UFL, lines 10 and 11 correspond to (1.2) and (1.3) respectively. Line 1 defines an abstract mesh of triangular cells in a 2-dimensional space with coordinates attached to each vertex (equivalent to a first-order Lagrange space). Setting `dim=3` would create a 2-dimensional surface in a 3-dimensional space. Setting the degree of the coordinate element higher than one would allow the triangular cells to be curved. Users of FEniCS and Firedrake typically create a concrete mesh object which extends the UFL class with mesh data that define a tesselation of a concrete physical domain. Line 2 sets up the function space $V = \hat{V}$ with P_2 elements on the mesh; we seek a solution in this space. Listing 1 specifies that κ and f are also from V.

Listing 1 Stationary heat equation in UFL

```
1  mesh = Mesh(VectorElement("Lagrange", triangle, 1, dim=2))
2  V = FunctionSpace(mesh, FiniteElement("Lagrange", triangle, 2))
3
4  u = TrialFunction(V)
5  v = TestFunction(V)
6
7  kappa = Coefficient(V)
8  f = Coefficient(V)
9
10 a = kappa*dot(grad(u), grad(v))*dx
11 L = f*v*dx
```

The UFL specification serves as the input of a form compiler. Form compilation can be understood as the *partial evaluation* of a multilinear form given a basis for each of its arguments. For example, the bilinear form a is compiled to the function $(x, \kappa) \mapsto \mathbf{A}$, and the linear form L to the function $(x, f) \mapsto \mathbf{b}$. Form compilers typically assume that the domain consists of a single cell; this makes the argument bases independent of actual physical domain up to a space transformation which is encapsulated in the coordinate coefficient x. This does not limit the applicability of form compilation since, given a tessellation $\mathcal{T}$ of the domain Ω, one can evaluate forms locally on each cell and assemble the contributions into the global tensors efficiently [19, §2.2].

1.2. A compiler pass. A compiler may be understood as a series of expression transformations, called *compiler passes*. A pass takes an expression and produces

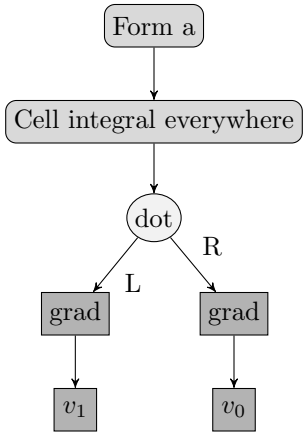

Figure 1: *Tree representation* of the form (1.6).

a new expression, implementing, for example, a simplification or an optimization, or translating the expression to a different language. A compiler may use several intermediate languages between the source and target language to decompose the translation process into several simpler passes, and especially to facilitate optimizations at the right level of abstraction.

It is common practice in form compilers to change the integral from physical to reference space. This enables, for example, the precomputation of basis functions at quadrature points. We use this transformation as an example for a compiler pass. Let us consider the Laplace operator

$$a(v,u) = \int_K \nabla u \cdot \nabla v \,\mathrm{d}x. \tag{1.6}$$

Let $x : \tilde{K} \to K$ be the mapping of coordinates from the reference cell to the physical cell, and let the Jacobian of this mapping be $J = \nabla x$. For a P_2 element $u(x(X)) = \tilde{u}(X)$, and likewise for v. The gradient transforms as $\nabla_x = J^{-T}\nabla$, and the integral needs scaling as $\mathrm{d}x = |J|\,\mathrm{d}X$. Thus we have

$$\int_K \nabla u \cdot \nabla v \,\mathrm{d}x = \int_{\tilde{K}} J^{-T}\nabla\tilde{u} \cdot J^{-T}\nabla\tilde{v}\,|J|\,\mathrm{d}X. \tag{1.7}$$

Figure 1 presents the input UFL form as an expression tree. By locally applying the above substitutions to the arguments, gradients and the integral, we obtain the expression graph in Figure 2. As one can see, J^{-T} is referenced twice, and J is referenced twice directly, or three times indirectly. A later pass may replace J with the evaluation of ∇x. It is therefore better to think of expressions as *directed acyclic graphs* (DAGs) rather than trees.

1.3. Facet integrals in discontinuous Galerkin methods. Discontinuous Galerkin methods connect cells through flux terms which are expressed as integrals over the *interior facets* of a mesh. Ølgaard, Logg, and Wells [33] extended the form language with interior facet integrals and provided an implementation in FFC, here

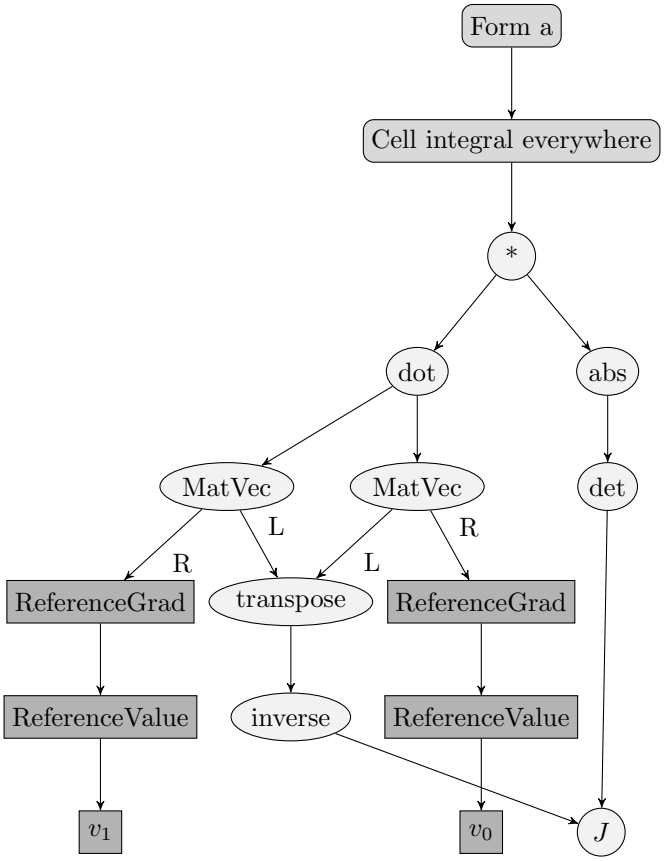

Figure 2: The Laplace operator after changing to reference space, equivalent to (1.7). The presence of two references (arrows) to J in place of replicating the J node make this a *directed acyclic graph* (DAG) rather than a tree.

we follow their description. Consider, for example, the bilinear form

$$a(v,u) = \int_S [\![u]\!][\![v]\!] \,\mathrm{d}s,$$

where $[\![v]\!]$ denotes the *jump* in the function value of v across the facet S. The jump is expressed as $[\![v]\!] = v^+ - v^-$, where v^+ and v^- are the function values as seen from the two cells K^+ and K^- incident to S, respectively (see Figure 3). While UFL provides utility functions such as `jump` and `avg` for convenience, its only additional primitive is *restriction*: quantities to be evaluated on interior facets must be restricted to either side of the facet. The two sides of a facet are arbitrarily marked as $+$ and $-$.

Let $\{\phi_i^+\}_{i=1}^n$ be the local finite element basis on K^+, and similarly let $\{\phi_i^-\}_{i=1}^n$ be the local finite element basis on K^-. Since interior facet integrals involve both cells,

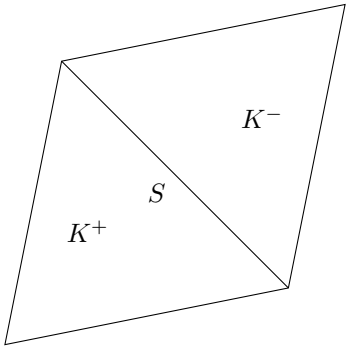

Figure 3: Two cells K^+ and K^- sharing a common facet S.

let $\{\phi_i\}_{i=1}^{2n}$ be a local basis on $K = K^+ \cup K^-$ by the following construction:

$$\phi_i(x) = \begin{cases} \phi_i^+(x) & \text{if } 1 \le i \le n \text{ and } x \in K^+, \\ 0 & \text{if } 1 \le i \le n \text{ and } x \in K^-, \\ 0 & \text{if } n+1 \le i \le 2n \text{ and } x \in K^+, \\ \phi_{i-n}^-(x) & \text{if } n+1 \le i \le 2n \text{ and } x \in K^-. \end{cases} \tag{1.8}$$

This construction makes the local tensor $\mathbf{A}_{ij} = a(\phi_i, \phi_j)$ a *block tensor*:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}^{(+,+)} & \mathbf{A}^{(+,-)} \\ \mathbf{A}^{(-,+)} & \mathbf{A}^{(-,-)} \end{bmatrix} \tag{1.9}$$

FFC, UFLACS and TSFC construct the interior facet tensor $\mathbf{A}$ by computing its subtensors. The subtensor $\mathbf{A}^{(+,+)}$ can be seen as the evaluation of a corresponding form $a^{(+,+)}$, and the same applies for the other subtensors. Section 2 presents a novel algorithm implemented in TSFC for the separation of the multilinear form into several forms corresponding to the subtensors of the local tensor.

1.4. Other contemporary form compilers. FFC contains several compilers for weak forms—they are called "representations". The *quadrature representation* [34] generates code that performs runtime quadrature, akin to what one would manually write. The *tensor representation* [19, 20] computes the local tensor as the contraction of a precomputable *reference tensor* with an element-dependent *geometry tensor*. This allows the elimination of the quadrature loop, which especially benefits higher-order schemes. However, due to the nature of this representation, the size of the reference tensor grows exponentially with the number of coefficients in the form, rendering this representation inefficient or even infeasible for a wide variety of problems. On the other hand, the quadrature representation suffers from an implementation limitation that expressions are seen as a tree rather than a DAG. This can dramatically increase the perceived size of the input expression (see Figure 4), which is detrimental to form compilation speed and generated code size. In case of non-affine geometries as well as for "complicated forms" — such as various formulations of hyperelastic models — the effect can be prohibitive.

UFLACS, which was integrated as another FFC representation, and has recently become the default implementation for most cases, has been developed to overcome

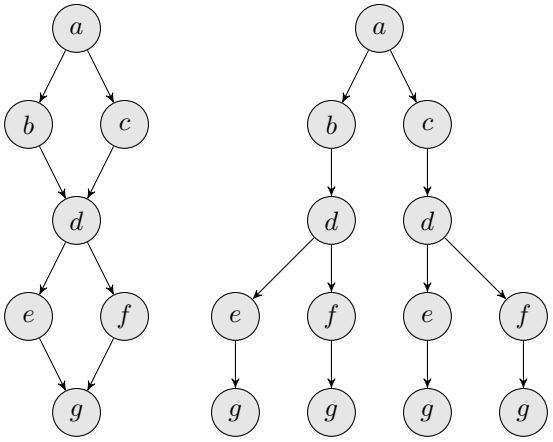

Figure 4: DAG representation of an expression with shared subexpressions (left), and a *tree view* of the same expression (right).

this limitation. However, to keep the internal data structures simple, UFLACS is eager to lower high-level constructs early in the form compilation pipeline. The high-level structure expressed in the input form, if preserved longer, could facilitate optimisations that are otherwise difficult to obtain. Section 2 provides further insight.

2. Preserving structure. Optimizations are best expressed at a specific level of abstraction. If the representation is too high level, the transformation may not be expressible at all, and if the representation is too low level, difficult analyses may be needed to recognize the optimization opportunity. It is therefore desirable to postpone representation lowering operations until the appropriate optimizations have been applied. There are two principal ways in which UFLACS lowers structure early, but TSFC does not: lowering tensor expressions to scalar expressions and *argument factorization*.

Argument factorization rewrites the integrand as a sum of products such that each product contains a term without arguments and for each argument a component of (a derivative of) that argument. This is always possible since the form is multilinear in its arguments. Lowering to scalars is a practical requirement: distributivity rules in vector and tensor algebra are numerous, and completeness of the implementation is necessary for correctness in all cases. The sum-of-product form makes it easy to separate the form into parts that correspond to the subtensors of the local tensor in interior facet integrals (see subsection 1.3). We also note that UFLACS applies argument factorization irrespective of the integral type. In some cases it turns out to be an effective optimization as argument-independent expressions are exposed and precomputed outside the inner loops. However, this approach is in general suboptimal. The transformation may require the expansion of products involving one or more arguments which affects an expression in two major ways: *a*) by increasing the operation count (which is usually only partially counteracted by the subsequent loop hoisting); and *b*) by destroying the common subexpressions that naturally arise when a problem is specified in a tensor algebra language.

To demonstrate what UFLACS does, let us consider the following interior facet

integral as an example:

$$\int_S [\![u]\!]_n \cdot \{\nabla v\} \, \mathrm{d}s \tag{2.1}$$

where $[\![u]\!]_n$ is the product of the jump of u with the facet normal n, and $\{\nabla v\}$ is the average of ∇v. The equivalent integrand with restrictions is

$$[\![u]\!]_n \cdot \{\nabla v\} = \frac{1}{2}\left(u^+ n^+ + u^- n^-\right) \cdot \left(\nabla v^+ + \nabla v^-\right). \tag{2.2}$$

n is of unit length, and always pointing out of the cell to which it is restricted. UFLACS first lowers the expression to scalars, in this case by expanding the dot product. Assuming a 2-dimensional space, this step yields

$$\begin{aligned}
[\![u]\!]_n \cdot \{\nabla v\} &= \frac{1}{2}\sum_{i=1}^{2}\left(u^+ n_i^+ + u^- n_i^-\right)\left(\left(\nabla v^+\right)_i + \left(\nabla v^-\right)_i\right) \\
&= \frac{1}{2}\Big[\left(u^+ n_1^+ + u^- n_1^-\right)\left(\left(\nabla v^+\right)_1 + \left(\nabla v^-\right)_1\right) \\
&\quad + \left(u^+ n_2^+ + u^- n_2^-\right)\left(\left(\nabla v^+\right)_2 + \left(\nabla v^-\right)_2\right)\Big].
\end{aligned}$$

Argument factorization is achieved through the application of distributivity rules:

$$\begin{aligned}
[\![u]\!]_n \cdot \{\nabla v\} = {} & \frac{1}{2}u^+ n_1^+ \left(\nabla v^+\right)_1 + \frac{1}{2}u^+ n_1^+ \left(\nabla v^-\right)_1 + \frac{1}{2}u^- n_1^- \left(\nabla v^+\right)_1 + \frac{1}{2}u^- n_1^- \left(\nabla v^-\right)_1 \\
& + \frac{1}{2}u^+ n_2^+ \left(\nabla v^+\right)_2 + \frac{1}{2}u^+ n_2^+ \left(\nabla v^-\right)_2 + \frac{1}{2}u^- n_2^- \left(\nabla v^+\right)_2 + \frac{1}{2}u^- n_2^- \left(\nabla v^-\right)_2.
\end{aligned}$$

At this point one can easily separate the form based on argument restrictions:

$$(+,+) \Rightarrow \frac{1}{2}u^+ n_1^+ \left(\nabla v^+\right)_1 + \frac{1}{2}u^+ n_2^+ \left(\nabla v^+\right)_2 \tag{2.3a}$$

$$(+,-) \Rightarrow \frac{1}{2}u^+ n_1^+ \left(\nabla v^-\right)_1 + \frac{1}{2}u^+ n_2^+ \left(\nabla v^-\right)_2 \tag{2.3b}$$

$$(-,+) \Rightarrow \frac{1}{2}u^- n_1^- \left(\nabla v^+\right)_1 + \frac{1}{2}u^- n_2^- \left(\nabla v^+\right)_2 \tag{2.3c}$$

$$(-,-) \Rightarrow \frac{1}{2}u^- n_1^- \left(\nabla v^-\right)_1 + \frac{1}{2}u^- n_2^- \left(\nabla v^-\right)_2. \tag{2.3d}$$

In contrast, TSFC separates the form into parts that correspond to the subtensors without applying argument factorization. This is achieved by simply exploiting the linearity of the form in its arguments. Since each instance of an argument is either positively or negatively restricted in the form, the following equality holds:

$$f(v^+, v^-) = f(v^+, 0) + f(0, v^-) \tag{2.4}$$

where f is the form interpreted as a function of two variables, the positively restricted and the negatively restricted argument. Equation (2.4) means that splitting the form based on its contributions to the subtensors of the interior facet tensor is as simple as substituting an appropriately shaped zero for any argument that has the "wrong" restriction. This approach also naturally generalizes to any number of arguments.

Applying this approach to (2.2) directly yields:

$$(+,+) \Rightarrow \frac{1}{2}\left(u^+n^+ + 0n^-\right)\cdot\left(\nabla v^+ + \nabla 0\right) = \frac{1}{2}u^+n^+\cdot\nabla v^+ \tag{2.5a}$$

$$(+,-) \Rightarrow \frac{1}{2}\left(u^+n^+ + 0n^-\right)\cdot\left(\nabla 0 + \nabla v^-\right) = \frac{1}{2}u^+n^+\cdot\nabla v^- \tag{2.5b}$$

$$(-,+) \Rightarrow \frac{1}{2}\left(0n^+ + u^-n^-\right)\cdot\left(\nabla v^+ + \nabla 0\right) = \frac{1}{2}u^-n^-\cdot\nabla v^+ \tag{2.5c}$$

$$(-,-) \Rightarrow \frac{1}{2}\left(0n^+ + u^-n^-\right)\cdot\left(\nabla 0 + \nabla v^-\right) = \frac{1}{2}u^-n^-\cdot\nabla v^-. \tag{2.5d}$$

Equations (2.3) and (2.5) are equivalent, but TSFC's approach preserves the tensor structure of the original expression, and does not require rearrangement. In general, the preserved structure can be exploited, for example, by COFFEE [31, 30], an optimizer for finite element loop nests. COFFEE explores a *transformation space* with the goal of minimizing the operation count of a kernel, and the sequence of transformations that UFLACS carries out is difficult to undo. This renders inaccessible to COFFEE a part of the transformation space, potentially eliminating the most efficient configurations.

Preserving the tensor structure of the UFL form presents further difficulties. Previous form compilers lower the form expression to scalars, and therefore only require code generation for the evaluation of finite element objects and geometric terms, while the expression itself is simply transcribed since the target language (typically C or C++) also supports scalar expressions. With tensor form expressions, code generation is also required for the translation of operations on tensor valued operands and for tensor contractions. Furthermore, the tensor nature of UFL expressions has two representations (see subsection 3.1). To tackle this complexity, we split compilation into two stages, hence the name: *Two-Stage Form Compiler* (TSFC). A tensor algebra language (GEM) is used as an intermediate representation, and the first stage lowers finite element objects and geometric terms to this language (section 3), while the second stage generates code to evaluate this tensor algebra expression (section 4). This architecture is sketched in Figure 5. We also expect our approach to be beneficial in the future for generating efficient code for higher-order elements by exploiting structure inside the elements and utilizing FInAT [18], a smarter library of finite elements.

3. First stage (UFL → GEM). GEM is a tensor algebra language very similar to UFL in its tensor algebra features, although it contains nothing specific to the finite element method. This approach works very well, because the evaluation of finite element objects and geometric terms is expressible using only tensor algebra.

3.1. Tensor algebra and index notation in UFL. The tensor nature of UFL expressions is represented as *shape* and *free indices*:

shape An ordered list of dimensions and their respective extent, e.g. (2, 2). A dimension is only identified by its position in the shape.

free indices An unordered set of dimensions where each dimension is identified by a symbolic index object. One might think of free indices as an "unrolled shape".

These traits are an integral part of any UFL expression. For example, let A be a 2×2 matrix, then A has shape (2, 2) and no free indices. $A_{1,1}$ has scalar shape and no free indices; $A_{i,j}$ has scalar shape and free indices i and j; and $A_{i,1}$ has scalar shape

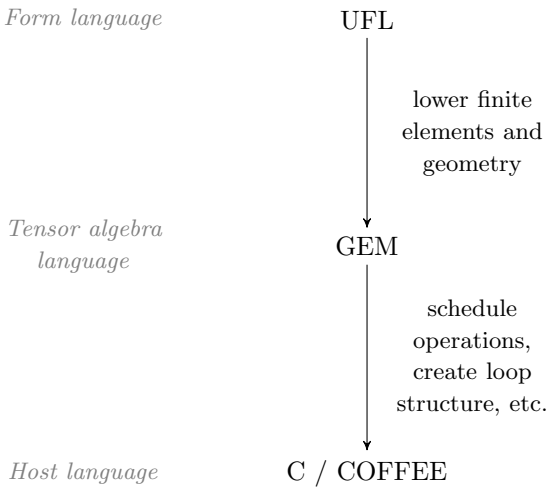

Figure 5: TSFC compiles forms in two stages. The first stage translates the evaluation of finite element objects to tensor algebra expressions. The second stage generates C code to evaluate tensor algebra expressions.

and free index i. UFL has four basic node types related to the tensor nature of its expressions:

- `Indexed`$(A, (\alpha_1, \alpha_2, \ldots, \alpha_r))$
  where A is an expression of rank r, and α is multi-index consisting of fixed indices and free indices. The result has scalar shape, but all the free indices of α become free indices of the result, i.e. `Indexed` can convert shape to free indices.
- `IndexSum`(e, i)
  Tensor contraction of the expression e over the free index i.
- `ComponentTensor`(e, α)
  where $\alpha := (\alpha_1, \alpha_2, \ldots, \alpha_k)$ is a multi-index consisting of free indices, and e is an expression with free indices $\alpha_1, \alpha_2, \ldots, \alpha_k$ (at least). These free indices are turned into shape, in the order as they appear in α.
- `ListTensor`$(e_1, e_2, \ldots, e_n)$
  where $e_1, e_2, \ldots, e_n$ are expressions of shape $(s_1, s_2, \ldots, s_r)$. The result is a concatenation of (the otherwise unrelated) expressions $e_1, e_2, \ldots, e_n$, with shape $(n, s_1, s_2, \ldots, s_r)$.

The index notation provides a low-level representation for many vector and matrix operations. For example, dot product, element-wise multiplication and outer product are all expressible using only multiplication of scalars and some of the above tensor operations:

- `dot(u, v)` ⇒ `IndexSum(u[i] * v[i], i)`
- `elem_mult(u, v)` ⇒ `ComponentTensor(u[i] * v[i], (i,))`
- `outer(u, v)` ⇒ `ComponentTensor(u[i] * v[j], (i, j))`

3.2. UFL preprocessing and modified terminals. TSFC relies on UFL to preprocess the form. The most important step in this procedure is to change the integral from physical to reference space. This involves scaling the integral and the

application of pullbacks on the functions (arguments and coefficients) of the form. For example, the Laplace operator

$$\int \nabla u \cdot \nabla v \,\mathrm{d}x, \tag{3.1}$$

where u and v are the trial and the test function respectively, becomes

$$\int J^{-T}\nabla\tilde{u} \cdot J^{-T}\nabla\tilde{v}\,|J|\,\mathrm{d}X \tag{3.2}$$

if v and u require no reference value transformation, as discussed in subsection 1.2. J is the Jacobian of the coordinate transformation. Note the change in the integral measure ($\mathrm{d}x$ versus $\mathrm{d}X$). After this transformation, UFL does various simplifications on the form: the Jacobian is replaced with the gradient of the spatial coordinates, determinants are expanded, divergence and curl are expressed with tensor algebra on gradients, and various products are expanded using the index notation. Subsection 3.1 demonstrated that lowering to the index notation introduces `ComponentTensor` nodes; however, to make this discussion easier to follow, we simplify most of them away, even though in reality they are only eliminated at a later stage (subsection 4.2). Finally, gradients and restrictions are propagated to the terminals. This reduces the number of language constructs that the form compiler needs to handle while preserving a rich language for the user. As a result of these operations, (3.2) becomes:

$$\begin{aligned}
d &:= \left(\frac{\partial x}{\partial X}\right)_{1,1}\left(\frac{\partial x}{\partial X}\right)_{2,2} - \left(\frac{\partial x}{\partial X}\right)_{1,2}\left(\frac{\partial x}{\partial X}\right)_{2,1} && \triangleright\, \det(J)\\
B &:= \begin{bmatrix} \left(\frac{\partial x}{\partial X}\right)_{2,2} & -\left(\frac{\partial x}{\partial X}\right)_{1,2} \\ -\left(\frac{\partial x}{\partial X}\right)_{2,1} & \left(\frac{\partial x}{\partial X}\right)_{1,1} \end{bmatrix} && \triangleright\, \mathrm{adj}(J)\\
K &:= \left]\frac{B_{i_1,i_2}}{d}\right[_{i_1,i_2} && \triangleright\, \mathrm{inv}(J)\\
&\int \sum_{i_5}\left(\sum_{i_3} K_{i_3,i_5}(\nabla\tilde{u})_{i_3}\right)\left(\sum_{i_4} K_{i_4,i_5}(\nabla\tilde{v})_{i_4}\right)|d|\,\mathrm{d}X.
\end{aligned} \tag{3.3}$$

where $]e[_\alpha$ is a notation for `ComponentTensor`(e, α). Equation (3.3) denotes a single expression, although some subexpressions are assigned names for simpler presentation.

A *terminal* is a node in an expression graph with no children. UFL terminals are: *a*) constant literals, *b*) coefficients, *c*) arguments, *d*) geometric quantities, and *e*) quadrature weight. Various *terminal modifiers* may wrap a terminal. TSFC only needs to handle the following terminal modifiers:

- `PositiveRestricted`, `NegativeRestricted`
  Restriction to one particular side of the facet in an interior facet integral.
- `ReferenceGrad`
  Gradient in the reference space, possibly applied multiple times.

A *modified terminal* is a terminal with a set of modifiers attached. In this set-up, the translation of modified terminals is independent of their context (surrounding outer expression), and the translation of the overall expression does not need to know more about the modified terminals than their shape and free indices. Since GEM intentionally mimics the tensor algebra features of UFL, the latter problem is trivial. We discuss the former in the next subsection.

3.3. Substitution of modified terminals. Currently, the substitution rules for modified terminals mimic the evaluation rules of FFC:

- **Arguments.** FIAT [17] provides the tabulation matrix that contains the value of each basis function at each quadrature point for the finite element of the argument. This becomes a compile-time constant matrix in GEM. This matrix is then indexed with the quadrature index and the corresponding argument index.
- **Coefficients.** We need a GEM expression that evaluates the coefficient at each quadrature point. The matrix-vector product of the element tabulation matrix with the local basis function coefficients gives exactly that. That vector for the cell is given as a kernel argument, which is represented as a run-time value in GEM.
- **Geometric quantities.** Because Firedrake represents the mesh coordinates as a finite element field, the Jacobian and further derivatives of the coordinate field are simply evaluated as coefficients. Other geometric terms typically appear in facet integrals (e.g. facet normal), and, as a consequence of UFL preprocessing, are expressed in reference space. They depend on the cell type, which is known at compile time, and often on the facet, which is provided as a kernel argument at run time. Thus the evaluation of a geometric term usually involves indexing into a compile-time constant tensor with the facet index.

Applying these evaluation rules, the modified terminals of (3.3) are replaced with the following expressions:

$$\nabla\tilde{v} \to \begin{bmatrix} E^{(1)}_{q,j} & E^{(2)}_{q,j} \end{bmatrix} \tag{3.4}$$

$$\nabla\tilde{u} \to \begin{bmatrix} E^{(1)}_{q,k} & E^{(2)}_{q,k} \end{bmatrix} \tag{3.5}$$

$$\frac{\partial x}{\partial X} \to \begin{bmatrix} \sum_r C^{(1,1)}_{q,r} c_r & \sum_r C^{(1,2)}_{q,r} c_r \\ \sum_r C^{(2,1)}_{q,r} c_r & \sum_r C^{(2,2)}_{q,r} c_r \end{bmatrix} \tag{3.6}$$

where $E^{(b)}$ is the tabulation matrix of the first derivate of the finite element of v in the b-th direction, $C^{(a,b)}$ is the tabulation matrix of the first derivate in the b-th direction of the a-th component of the coordinate element, and c is the vector of the local basis function coefficients of the coordinates. This general treatment of the Jacobian also works for non-simplex cells and higher-order geometries. We will discuss optimizations for affine simplices in section 5. Since indexing a tensor with fixed indices immediately selects the appropriate entry, these substitutions yield the following integrand expression:

$$J^{(1,1)} := \sum_r C^{(1,1)}_{q,r} c_r$$
$$J^{(1,2)} := \sum_r C^{(1,2)}_{q,r} c_r$$
$$J^{(2,1)} := \sum_r C^{(2,1)}_{q,r} c_r$$
$$J^{(2,2)} := \sum_r C^{(2,2)}_{q,r} c_r$$

$$d := J^{(1,1)} J^{(2,2)} - J^{(1,2)} J^{(2,1)}$$
$$B := \begin{bmatrix} J^{(2,2)} & -J^{(1,2)} \\ -J^{(2,1)} & J^{(1,1)} \end{bmatrix}$$
$$K := \left] \frac{B_{i_1,i_2}}{d} \right[_{i_1,i_2}$$

$$\int \sum_{i_5} \left(\sum_{i_3} K_{i_3,i_5} \left[E^{(1)}_{q,k} \quad E^{(2)}_{q,k} \right]_{i_3} \right) \left(\sum_{i_4} K_{i_4,i_5} \left[E^{(1)}_{q,j} \quad E^{(2)}_{q,j} \right]_{i_4} \right) |d| \, \mathrm{d}X.$$

Aggregating the integrand values for each quadrature point with quadrature weight w_q gives the GEM representation of (3.1) as

$$\sum_q w_q \, |d| \sum_{i_5} \left(\sum_{i_3} K_{i_3,i_5} \left[E^{(1)}_{q,k} \quad E^{(2)}_{q,k} \right]_{i_3} \right) \left(\sum_{i_4} K_{i_4,i_5} \left[E^{(1)}_{q,j} \quad E^{(2)}_{q,j} \right]_{i_4} \right). \tag{3.7}$$

This representation only contains tensor algebra operators applied to compile-time constant tensors and a run-time argument to the local kernel. All the finite element and geometric terms have been lowered, the first stage of form compilation is complete.

3.4. Intermediate representation (GEM). Since GEM mimics the tensor algebra features of UFL and lacks other features, such as finite element objects and geometric quantities, one may rightly ask why the tensor algebra subset of UFL could not be adopted as the intermediate representation. The reason for the creation of GEM is to relax some of the constraints on *free indices* which are intrinsic in UFL.

Free indices are just an "unrolled shape" in UFL, that may be "rolled back" any time. For any operation, free indices of the operands are subject to the same constraints that apply to shape. For example, tensors must have equal shape for addition, consequently, they also must have the same free indices. GEM interprets free indices differently. A free index in GEM simply means that the value of the expression depends on that index. For example, let u be a 3-dimensional vector (no free indices). u_i and u_j are scalars with free indices i and j, respectively. So far, it is identical to UFL. However, in GEM $u_i + u_j$ is just an expression that depends on both i and j, that is a scalar with free indices i and j, while in UFL the same expression is illegal because the free indices do not match: the addition of a 3-dimensional vector with itself cannot produce a 3×3 matrix.

While the enforcement of strict shape checking is valuable for preventing user errors in a weak form, this strictness becomes an obstacle when the modified terminals are replaced with their respective evaluation rules. For example, we replaced $\nabla \tilde{v}$ with $\left[E^{(1)}_{q,j} \quad E^{(2)}_{q,j} \right]$ in subsection 3.3. In UFL this would mean replacing a vector with a third-order tensor, which yields an illegal expression in the general case. In GEM it just replaces a vector with another vector which depends on q (quadrature index) and j (argument index). There is eventually a summation over q, and the dependence on j gives the components of the element tensor. This difference in free indices poses no difficulty for the UFL $\rightarrow$ GEM translation since GEM is strictly more permissive.

4. Second stage (GEM $\rightarrow$ C). This section discusses the second stage of form compilation, which is the translation of GEM to C, a programming language with explicit loops and only scalar expressions.

4.1. Overview of GEM. GEM was intentionally designed to be very similar to UFL in its tensor algebra features. In common with UFL, the tensor nature of expressions can be expressed as *shape* or *free indices* (see subsection 3.1). Most supported operations are naturally scalar, such as addition and multiplication, but there are a few tensor operations as well, such as tensor contraction (called `IndexSum`). Here we give a summary of all GEM nodes:

- **Terminals:**

  - `Literal` (tensor literal)
  - `Zero` (all-zero tensor)
  - `Identity` (identity matrix)
  - `Variable` (run-time value, for kernel arguments)
- **Scalar operations:**
  - *Binary operators:* `Sum`, `Product`, `Division`, `Power`, `MinValue`, `MaxValue`
  - *Unary operators:* `MathFunction` (e.g. sin, cos)
  - `Comparison` ($>$, $\geq$, $=$, $\neq$, $<$, $\leq$): compares numbers, returns Boolean
  - *Logical operators:* `LogicalAnd`, `LogicalOr`, `LogicalNot`
  - `Conditional`: selects between a "true" and a "false" expression based on a Boolean valued *condition*
- **Index types:**
  - `int` (fixed index)
  - `Index` (free index): creates a loop at code generation
  - `VariableIndex` (unknown fixed index): index value only known at run-time, e.g. facet number
- **Tensor nodes:** `Indexed`, `ComponentTensor`, `IndexSum`, `ListTensor`
  See subsection 3.1 for details.

4.2. Elimination of `ComponentTensor` nodes. The duplicity of *shape* and *free indices* would noticeably complicate the process of code generation. To avoid that, we eliminate non-scalar shape in non-terminal nodes by removing all `ComponentTensor` nodes using a rule that is analogous to *beta reduction*:

$$\texttt{Indexed}(\texttt{ComponentTensor}(e, \alpha), \beta) \to e|_{\alpha \to \beta} \tag{4.1}$$

where e is an expression with scalar shape, α is a multi-index containing distinct free indices of e, and β is a multi-index of the same rank as α. $e|_{\alpha \to \beta}$ denotes a substitution of indices in e, where α_i is replaced with β_i for all i. The rule replaces indexing into a `ComponentTensor` by a substitution of the indices.

A `ComponentTensor` node must be immediately inside an `Indexed` node or at the root of the expression because `Indexed` is the only GEM operator that accepts non-scalar shape. Since the UFL $\to$ GEM stage produces a GEM expression with scalar shape and the argument indices as free indices, the application of (4.1) removes all `ComponentTensor`s from the intermediate representation. The downside of this approach is the possibility of code duplication when different substitutions are applied on the same subexpression.

To demonstrate this, we continue with our Laplace operator example. To simplify our presentation in subsection 3.2, most `ComponentTensor` nodes were eliminated except for K defined in (3.3). Eliminating K in (3.7) gives

$$\sum_q w_q \, |d| \sum_{i_5} \left(\sum_{i_3} \frac{B_{i_3,i_5}}{d} \begin{bmatrix} E^{(1)}_{q,k} & E^{(2)}_{q,k} \end{bmatrix}_{i_3} \right) \left(\sum_{i_4} \frac{B_{i_4,i_5}}{d} \begin{bmatrix} E^{(1)}_{q,j} & E^{(2)}_{q,j} \end{bmatrix}_{i_4} \right). \tag{4.2}$$

As one can see, this step duplicates the element-wise division of the adjugate matrix by the determinant.

4.3. Code generation. Since `ComponentTensor` nodes are eliminated, we can now restrict our attention to the index notation. A simple code generation approach for the evaluation of such tensor expressions is to introduce a *temporary variable* and generate a pair of *perfect loop nests* (Figure 6a) for each tensor contraction

```
for all i do
    for all j do
        d[i, j] ← (a[i] + b[i]) ∗ c[j]
```

(a) A *perfect loop nest* of two loops: the inner loop is perfectly nested in the outer loop since the outer loop contains no other statements than the inner loop.

```
for all i do
    t[i] ← a[i] + b[i]
    for all j do
        d[i, j] ← t[i] ∗ c[j]
```

(b) The result of hoisting operations from the inner loop; an *imperfect* loop nest.

Figure 6: An example of *loop hoisting.* The calculation $a[i] + b[i]$ is independent of the inner loop, thus it need not be repeated for all j.

(`IndexSum`). The first loop nest initializes the temporary, and the second accumulates the result. Consequently, the first loop nest contains all the free indices of the `IndexSum` expression, while the second loop nest has one more index, the index being summed over. For example, applying this approach to (4.2), we introduce temporaries $J^{(1,1)}$, $J^{(1,2)}$, $J^{(2,1)}$, $J^{(2,2)}$, t_1, t_2, t_3 and A. The evaluation code is presented in Algorithm 1, which also adds loop nests for "defining" `ListTensor` expressions, such as adjungate matrix B, $l_1 = \begin{bmatrix} E^{(1)}_{q,k} & E^{(2)}_{q,k} \end{bmatrix}$, and $l_2 = \begin{bmatrix} E^{(1)}_{q,j} & E^{(2)}_{q,j} \end{bmatrix}$.

Algorithm 1 Simple code generation for the expression (4.2).

```
for all q do
    J^(1,1)[q] ← 0
for all q, r do
    J^(1,1)[q] += C^(1,1)_{q,r} ∗ c_r
for all q do
    J^(1,2)[q] ← 0
for all q, r do
    J^(1,2)[q] += C^(1,2)_{q,r} ∗ c_r
for all q do
    J^(2,1)[q] ← 0
for all q, r do
    J^(2,1)[q] += C^(2,1)_{q,r} ∗ c_r
for all q do
    J^(2,2)[q] ← 0
for all q, r do
    J^(2,2)[q] += C^(2,2)_{q,r} ∗ c_r
for all q do
    B[q, ...] ← [ J^(2,2)[q]   −J^(1,2)[q] ]
                [ −J^(2,1)[q]  J^(1,1)[q]  ]
for all q, k do
    l_1[q, k, ...] ← [ E^(1)_{q,k}  E^(2)_{q,k} ]
for all i_5, q, k do
    t_1[i_5, q, k] ← 0
for all i_3, i_5, q, k do
    t_1[i_5, q, k] += B[q, i_3, i_5] ÷ (J^(1,1)[q] ∗
    J^(2,2)[q] − J^(1,2)[q] ∗ J^(2,1)[q]) ∗ l_1[q, k, i_3]
for all q, j do
    l_2[q, j, ...] ← [ E^(1)_{q,j}  E^(2)_{q,j} ]
for all i_5, q, j do
    t_2[i_5, q, j] ← 0
for all i_4, i_5, q, j do
    t_2[i_5, q, j] += B[q, i_4, i_5] ÷ (J^(1,1)[q] ∗
    J^(2,2)[q] − J^(1,2)[q] ∗ J^(2,1)[q]) ∗ l_2[q, j, i_4]
for all q, j, k do
    t_3[q, j, k] ← 0
for all i_5, q, j, k do
    t_3[q, j, k] += t_1[i_5, q, k] ∗ t_2[i_5, q, j]
for all j, k do
    A[j, k] ← 0
for all j, k, q do
    A[j, k] += w_q ∗ abs(J^(1,1)[q] ∗
    J^(2,2)[q]−J^(1,2)[q]∗J^(2,1)[q])∗t_3[q, j, k]
```

While this simple approach produces correct code, introducing temporaries merely for tensor contractions may cause too much *inlining*. In particular, shared subexpressions other than tensor contractions are still expanded, and there is also room for *loop hoisting* (Figure 6). Therefore, we also introduce temporaries to evaluate subexpressions that *a*) have a *reference count* greater than one, or *b*) have a different set of *free indices* than their unique parent. In our example, the determinant of the Jacobian is an instance of *a*), and $w_q * \mathrm{abs}(\ldots)$ in the final loop nest is an instance of *b*). So we introduce temporaries d and t_4 respectively. The result of this is Algorithm 2.

Algorithm 2 Introducing further temporaries in Algorithm 1.

```
... skipped ...
for all q do                                          ▷ Common subexpression
    d[q] = J^(1,1)[q] * J^(2,2)[q] − J^(1,2)[q] * J^(2,1)[q]
for all i3, i5, q, k do
    t1[i5, q, k] += B[q, i3, i5] ÷ d[q] * l1[q, k, i3]
for all q, j do
    l2[q, j, ...] ← [E^(1)_{q,j}  E^(2)_{q,j}]
for all i5, q, j do
    t2[i5, q, j] ← 0
for all i4, i5, q, j do
    t2[i5, q, j] += B[q, i4, i5] ÷ d[q] * l2[q, j, i4]
for all q, j, k do
    t3[q, j, k] ← 0
for all i5, q, j, k do
    t3[q, j, k] += t1[i5, q, k] * t2[i5, q, j]
for all j, k do
    A[j, k] ← 0
for all q do                                          ▷ Hoisted computation
    t4[q] = w_q * abs(d[q])
for all j, k, q do
    A[j, k] += t4[q] * t3[q, j, k]
```

Finally, we apply *loop fusion* to reduce the size of the temporaries (see Figure 7). Here we use a simple and fast algorithm. First, we construct an ordering of all free indices that appear in the GEM expression, and apply this ordering to each (perfect) loop nest. Based on our knowledge about the structure of finite element assembly kernels, and in common with FFC, we choose the *quadrature index* to be the first (outermost), followed by the *argument indices* in order, and then by all other indices. A possible ordering that we use for our example is $(q, j, k, r, i_3, i_4, i_5)$. This means a number of *loop interchanges* (Figure 8) in Algorithm 2. Considering dependencies, we also take a *topological ordering* of the loop nests that heuristically tries to maximize loop fusion (subsection 4.4). For our example, this results in Algorithm 3. Finally, we apply maximal loop fusion without further reordering. The fused algorithm is Algorithm 4. Note that many of the temporaries are now scalar, or at least have a reduced tensor order.

Listing 2 is a straighforward transcription of Algorithm 4 to C code. The definitions of compile-time constant tensors $C^{(1,1)}$, $C^{(1,2)}$, $C^{(2,1)}$, $C^{(2,2)}$, w, $E^{(1)}$, and $E^{(2)}$

```
for all i do
    c[i] ← a[i] + b[i]
for all i do
    d[i] ← c[i] * c[i]
```

(a) Sequence of two loops

```
for all i do
    c[i] ← a[i] + b[i]
    d[i] ← c[i] * c[i]
```

(b) After loop fusion

```
for all i do
    c ← a[i] + b[i]
    d[i] ← c * c
```

(c) After shape reduction

Figure 7: An example of a pair of loops before and after *loop fusion*. If c is not needed later, its shape can be reduced after fusion.

```
for all i do
    for all j do
        c[i, j] ← a[i] * b[j]
```

```
for all j do
    for all i do
        c[i, j] ← a[i] * b[j]
```

Figure 8: An example of *loop interchange*: the inner and outer loops are swapped. Loop interchange is only applicable to a perfect loop nest (see Figure 6a).

Algorithm 3 Reordering loops in Algorithm 2.

```
for all j, k do
    A[j, k] ← 0
for all q do
    J^(1,1)[q] ← 0
for all q do
    J^(1,2)[q] ← 0
for all q do
    J^(2,1)[q] ← 0
for all q do
    J^(2,2)[q] ← 0
for all q, r do
    J^(1,1)[q] += C^(1,1)_{q,r} * c_r
for all q, r do
    J^(1,2)[q] += C^(1,2)_{q,r} * c_r
for all q, r do
    J^(2,1)[q] += C^(2,1)_{q,r} * c_r
for all q, r do
    J^(2,2)[q] += C^(2,2)_{q,r} * c_r
for all q do
    B[q, ...] ← [ J^(2,2)[q]   -J^(1,2)[q] ]
                [ -J^(2,1)[q]   J^(1,1)[q] ]
for all q do
    d[q] = J^(1,1)[q] * J^(2,2)[q] - J^(1,2)[q] * J^(2,1)[q]
for all q, k do
    l_1[q, k, ...] ← [ E^(1)_{q,k}   E^(2)_{q,k} ]
for all q, k, i_5 do
    t_1[q, k, i_5] ← 0
for all q, k, i_3, i_5 do
    t_1[q, k, i_5] += B[q, i_3, i_5] ÷ d[q] * l_1[q, k, i_3]
for all q do
    t_4[q] = w_q * abs(d[q])
for all q, j do
    l_2[q, j, ...] ← [ E^(1)_{q,j}   E^(2)_{q,j} ]
for all q, j, i_5 do
    t_2[q, j, i_5] ← 0
for all q, j, i_4, i_5 do
    t_2[q, j, i_5] += B[q, i_4, i_5] ÷ d[q] * l_2[q, j, i_4]
for all q, j, k do
    t_3[q, j, k] ← 0
for all q, j, k, i_5 do
    t_3[q, j, k] += t_1[q, k, i_5] * t_2[q, j, i_5]
for all q, j, k do
    A[j, k] += t_4[q] * t_3[q, j, k]
```

Algorithm 4 The result of loop fusion on Algorithm 3.

1: **for all** j, k **do**
2: $\quad A[j,k] \leftarrow 0$
3: **for all** q **do**
4: $\quad J^{(1,1)} \leftarrow 0$
5: $\quad J^{(1,2)} \leftarrow 0$
6: $\quad J^{(2,1)} \leftarrow 0$
7: $\quad J^{(2,2)} \leftarrow 0$
8: $\quad$ **for all** r **do**
9: $\quad\quad J^{(1,1)} \mathrel{+}= C^{(1,1)}_{q,r} * c_r$
10: $\quad\quad J^{(1,2)} \mathrel{+}= C^{(1,2)}_{q,r} * c_r$
11: $\quad\quad J^{(2,1)} \mathrel{+}= C^{(2,1)}_{q,r} * c_r$
12: $\quad\quad J^{(2,2)} \mathrel{+}= C^{(2,2)}_{q,r} * c_r$
13: $\quad B = \begin{bmatrix} J^{(2,2)} & -J^{(1,2)} \\ -J^{(2,1)} & J^{(1,1)} \end{bmatrix}$
14: $\quad d = J^{(1,1)} * J^{(2,2)} - J^{(1,2)} * J^{(2,1)}$
15: $\quad$ **for all** k **do**
16: $\quad\quad l_1 \leftarrow \begin{bmatrix} E^{(1)}_{q,k} & E^{(2)}_{q,k} \end{bmatrix}$
17: $\quad\quad$ **for all** i_5 **do**
18: $\quad\quad\quad t_1[k, i_5] \leftarrow 0$
19: $\quad\quad$ **for all** i_3, i_5 **do**
20: $\quad\quad\quad t_1[k, i_5] \mathrel{+}= B[i_3, i_5] \div d * l_1[i_3]$
21: $\quad t_4 = w_q * \mathrm{abs}(d)$
22: $\quad$ **for all** j **do**
23: $\quad\quad l_2 \leftarrow \begin{bmatrix} E^{(1)}_{q,j} & E^{(2)}_{q,j} \end{bmatrix}$
24: $\quad\quad$ **for all** i_5 **do**
25: $\quad\quad\quad t_2[i_5] \leftarrow 0$
26: $\quad\quad$ **for all** i_4, i_5 **do**
27: $\quad\quad\quad t_2[i_5] \mathrel{+}= B[i_4, i_5] \div d * l_2[i_4]$
28: $\quad\quad$ **for all** k **do**
29: $\quad\quad\quad t_3 \leftarrow 0$
30: $\quad\quad\quad$ **for all** i_5 **do**
31: $\quad\quad\quad\quad t_3 \mathrel{+}= t_1[k, i_5] * t_2[i_5]$
32: $\quad\quad\quad A[j,k] \mathrel{+}= t_4 * t_3$

is prepended to the function body. The element tensor A and the local coordinate coefficients vector c are defined as kernel arguments. Lines 10–14 may be omitted if the caller clears `A`.

4.4. Topological orderings of loop nests. The GEM DAG describes the dependencies of each node in that DAG. The nodes can be evaluated in any order which respects those dependencies. That is, if there is an edge from node a to node b then node b must be evaluated before node a. Any ordering which respects the constraints imposed by the DAG edges is called a *topological ordering*, and the process of generating such an ordering is called *topological sorting*.

Topological sorting is a well-understood problem [9, §22.4]. Every DAG admits a

Listing 2 Local assembly kernel for $\int \nabla u \cdot \nabla v \, \mathrm{d}x$ in C.

```
static inline void kernel(double A[4][4], const double c[8])
{
    static const double C11[4][8] = { ... };
    static const double C12[4][8] = { ... };
    static const double C21[4][8] = { ... };
    static const double C22[4][8] = { ... };
    static const double w[4] = { ... };
    static const double E1[4][4] = { ... };
    static const double E2[4][4] = { ... };
    for (int j = 0; j < 4; j++) {
        for (int k = 0; k < 4; k++) {
            A[j][k] = 0;
        }
    }
    for (int q = 0; q < 4; q++) {
        double J11 = 0;
        double J12 = 0;
        double J21 = 0;
        double J22 = 0;
        for (int r = 0; r < 8; r++) {
            J11 += C11[q][r]*c[r];
            J12 += C12[q][r]*c[r];
            J21 += C21[q][r]*c[r];
            J22 += C22[q][r]*c[r];
        }
        double B[2][2] = { { J22, -J12 }, { -J21, J11 } };
        double d = J11*J22 - J12*J21;
        double t1[4][2];
        for (int k = 0; k < 4; k++) {
            double l1[2] = { E1[q][k], E2[q][k] };
            for (int i_5 = 0; i_5 < 2; i_5++) {
                t1[k][i_5] = 0;
            }
            for (int i_3 = 0; i_3 < 2; i_3++) {
                for (int i_5 = 0; i_5 < 2; i_5++) {
                    t1[k][i_5] += (B[i_3][i_5]/d)*l1[i_3];
                }
            }
        }
        double t4 = w[q]*fabs(d);
        for (int j = 0; j < 4; j++) {
            double l2[2] = { E1[q][j], E2[q][j] };
            double t2[2];
            for (int i_5 = 0; i_5 < 2; i_5++) {
                t2[i_5] = 0;
            }
            for (int i_4 = 0; i_4 < 2; i_4++) {
                for (int i_5 = 0; i_5 < 2; i_5++) {
                    t2[i_5] += (B[i_4][i_5]/d)*l2[i_4];
                }
            }
            for (int k = 0; k < 4; k++) {
                double t3 = 0;
                for (int i_5 = 0; i_5 < 2; i_5++) {
                    t3 += t1[k][i_5]*t2[i_5];
                }
            A[j][k] += t4*t3;
        }
    }
}
```

topological ordering, but this is not in general unique. For correct code generation as described in subsection 4.3, any topological ordering suffices. However, the choice of ordering determines which loops can be fused, and thus how much memory is saved. Reducing the *working set size*[1] can help to fit into faster memory, thus also reducing execution time.

We use a variant of Kahn's algorithm [16] [21, §2.2.3, Algorithm T] with heuristics to maximize loop fusion. Algorithm 5 presents Kahn's algorithm for topological sorting. Line 6 is a point of ambiguity, since S can have more than one element. Any choice will yield a correct, but different, topological ordering, so our heuristics are about choosing "wisely".

Algorithm 5 Kahn's algorithm for *topological sorting*

```
 1: function TOPOLOGICAL-SORT(G)
 2:     refcount ← in-degree for each node in G
 3:     L ← [ ]
 4:     S ← {u | refcount[u] = 0}                    ▷ root nodes of G
 5:     while S is non-empty do
 6:         u ← CHOOSE-ELEMENT(S)
 7:         S ← S − {u}
 8:         insert u to the front of L
 9:         for all v ∈ children[u] do
10:             DECREMENT(refcount[v])
11:             if refcount[v] = 0 then
12:                 S ← S ∪ {v}
13:     return L
```

Loop fusion occurs between two consecutive loop nests for all common outer loops. We maximize this at every choice in a greedy manner. The number of loops in the previously chosen loop nest, however, poses an upper limit on the number of loops that can be fused as a result of the current choice. For example, if the previous choice was a loop nest with indices $(q,)$, this rule so far expresses no preference between $(q,)$ and (q, r) for the next choice, because only the q loop can be fused in both cases. Therefore we further refine our heuristic to prefer those loop nests with exactly the same loops as the previously chosen loop nest. Delaying "entry" into further inner loops, is a heuristic designed in the hope that more loop nests with those indices will satisfy the dependency requirements once we have no other option but to *do* "enter" those further inner loops, thus enabling more loop fusion later.

Algorithm 6 shows our version of the CHOOSE-ELEMENT function. We maintain the ordered loop indices of the previously chosen loop nest in *cursor*, which is initialized to () at the beginning of a topological sort. We can assume that S is non-empty. First, we try to choose an element with the same loop indices as the previous one (lines 3–5). If that is not possible, we try to choose an element such that *cursor* is a prefix of its loop indices (lines 6–10), and we update *cursor* accordingly. If that is not possible either, we drop the last (innermost) index from *cursor* and repeat. Before this step *cursor* must have had at least one index, otherwise $S_2 = S \neq \varnothing$, so the function would have returned in line 10.

Figure 9 shows the *dependency graph* of the loop nests in Algorithm 2, along with the solution our algorithm yields. The ordering precisely corresponds to what we have

[1] The amount of memory required to carry out a sequence of operations.

Algorithm 6 Heuristic to maximize loop fusion

```
1: function CHOOSE-ELEMENT(S)
2:     while true do
3:         S_1 = {u ∈ S | loop_indices[u] = cursor}
4:         if S_1 is non-empty then
5:             return any element from S_1
6:         S_2 = {u ∈ S | cursor is a prefix of loop_indices[u]}
7:         if S_2 is non-empty then
8:             u ← any element from S_2
9:             cursor ← loop_indices[u]
10:            return u
11:        remove last index from cursor
```

seen in Algorithm 3.

4.5. Related work. Since code generation for tensor algebra expressions is not inherently specific to the finite element method, the related work is broad in scope. Various tensor algebra libraries and domain-specific languages [7, 11, 36, 38] have been developed for quantum chemistry computations. However, while the *working set size* of a finite element local assembly kernel rarely exceeds a megabyte, the tensors in quantum chemistry computations are typically huge, even reaching petabytes in size. Consequently, the optimized implementation of individual tensor operations on distributed-memory parallel computers is usually the major concern, as is the case with SIAL [36], libtensor [11], and CTF [38]. The *Tensor Contraction Engine* (TCE) [7] goes further, and also considers optimizations on the whole tensor algebra expression, which could be applicable to our case as well. Some of these ideas were also applied to do efficient tensor contractions on GPUs [32].

Algebraic transformations on tensor algebra expressions can yield equivalent formulations that require many fewer operations to compute. Lam *et al.* [26] first studied the problem of *single-term optimization*, also known as *strength reduction*: the operation-minimal evaluation of tensor contractions over the product of several tensors with various indices. They prove that the problem is NP-complete, but they also provide a pruning search strategy that is efficient in practice, since a single term typically consists of only a limited number of tensors. Kschischang *et al.* [22] show that such a single term can be characterized by a *factor graph*. If the factor graph is *cycle-free* then it straightforwardly yields the unique optimal factorization. Otherwise different factorizations can be produced by various transformations that break the cycles. Considering the sum of single terms, Hartono *et al.* [13] achieved further improvements over single-term optimization by considering various factorizations of the whole expression. Since the space of different factorizations is huge, they relied on various search techniques. Exploiting *common subexpression elimination* during the search resulted in further gains [12]. We expect to benefit from these results in the future, for example, when applying sum factorization to forms defined over tensor product elements. At the moment TSFC primarily relies on COFFEE to provide operation minimization (see subsection 5.5).

Since tensors in quantum chemistry computations are huge, optimizing the memory requirement is of great importance so that computations can fit in the main memory. While this is not an issue for finite element local assembly, fitting the working set into the cache can have an important performance impact. The evaluation

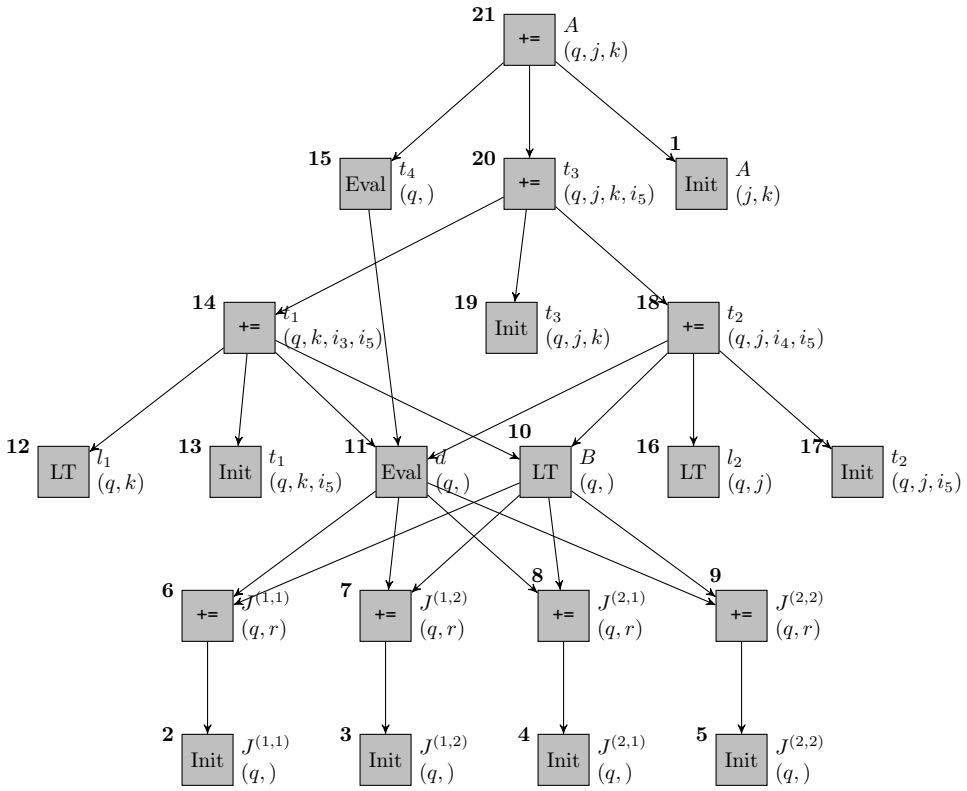

Figure 9: *Dependency graph* of the loop nests of Algorithm 2. At the right of each node are the name of the *temporary variable* and the ordered loop indices of the loop nest. The bold numbers at the upper left of each node represent a valid ordering of these operations.

of tensor expressions in a scalar programming language such as C requires the introduction of *array temporaries* which may be indexed. The order in which these temporaries are produced and consumed affects the working set size of the kernel. Lam *et al.* [23, 25] give an algorithm for the evaluation order of nodes of an expression tree that minimizes the maximal total size of live temporaries. Their work can be understood as a generalization of the Sethi-Ullman algorithm [37] where nodes are not assumed to have the same size, instead each node can have an arbitrary size. Since tensor valued temporaries are both produced and consumed by loops, *loop fusion* can eliminate dimensions of these temporaries, thus reducing the required memory, as we have demonstrated in subsection 4.3. Lam *et al.* [24] formalize the legality of loop fusions over expression trees and their effect on the memory requirement. They give an algorithm for finding the memory-optimal set of loop fusions based on dynamic programming and pruning, with an exponential complexity in the number of index variables. Allam *et al.* [1] propose an alternative algorithm that reduces this problem to an *integer linear programming* formulation. Cociorva *et al.* [8] studied space-time trade-off optimization in this context, by allowing some recomputations for further loop fusion opportunities. In TSFC we have implemented a fast algorithm

that may produce suboptimal results (subsections 4.3 and 4.4), rather than striving for optimality at the cost of high algorithmic complexity [24, 1],

5. Optimizations. This section discusses various optimizations in TSFC which are not essential for the understanding of the overall form compilation process, and whose proper effects are often only understood in composition with other optimizations at various points of the form compilation pipeline.

5.1. Constant folding for zero tensors. *Constant folding* means replacing computations whose all operands are known at compile time with the result of the computation. This is a standard technique in modern optimizing compilers. TSFC implements a limited version of it at the GEM level: only computations involving *zero tensors* are folded. This limited version can still yield significant benefits since the application of $0 \cdot X = 0$ can eliminate an arbitrarily complex expression X. Zero tensors can arise, for example, as tabulation matrices for higher derivatives than the degree of the finite element.

5.2. Cellwise constant quantities. We saw in subsection 3.3 that the evaluation of arguments and coefficients involves a tabulation matrix $M_{q,r}$ that contains the value of each basis function at each quadrature point. When an argument or a coefficient, or more commonly a derivative of it, is cellwise constant, then evaluation need not happen for every quadrature point. So instead of the matrix $M_{q,r}$ we create a vector M_r which does not have the quadrature index as a free index. This change in combination with loop hoisting (see subsection 4.3) will hoist the evaluation of all cellwise constant subexpressions out of the quadrature loop.

5.3. Fast Jacobian evaluation for affine cells. The Jacobian of the coordinate transformation appears in virtually every form after preprocessing, even the simplest ones. In those cases an inefficient Jacobian evaluation may likely become the dominant cost of assembly. Consequently, FEniCS has, and Firedrake used to have, efficient hand-written code snippets for Jacobian evaluation on affine simplices. As described in subsection 3.3, we now implement Jacobian evaluation as coefficient evaluation. This allows us to handle arbitrary order geometries with a single code path, but it also implies that each entry of the Jacobian is evaluated as a matrix-vector product. For affine cells, the Jacobian is cellwise constant, so this reduces to a dot product of two vectors (subsection 5.2). The code snippets evaluate each Jacobian entry with a single subtraction.

Since both approaches are correct and thus yield identical results, it follows immediately that the *tabulation vector* must have precisely two nonzero entries which are 1 and -1. Therefore, we introduced another optimization rule which applies to all coefficient evaluation: if the quantity is cellwise constant and the tabulation vector has at most two nonzero entries, then we expand the dot product that evaluates the coefficient. This, of course, composes with constant folding (subsection 5.1), so the multiplications with zeros vanish, and we obtain a formulation that is performance equivalent to the code snippets.

5.4. Unrolling short `IndexSum` loops. In most cases, unrolling short loops, which are typically geometric loops, improves performance. When this optimization is applied, the generated code for the Laplace operator becomes as in Algorithm 7. It also negates the code duplication around Jacobian inverses that was caused by the *beta expansion* step (subsection 4.2). This optimization is enabled by default in TSFC. However, as demonstrated, it is completely optional, and can be delayed to run after those optimizations that benefit from the tensor structure of the source expression.

Algorithm 7 Generated code when unrolling applied, equivalent to Algorithm 4.

for all j, k **do**
  $A[j,k] \leftarrow 0$
for all q **do**
  $J^{(1,1)} \leftarrow 0$
  $J^{(1,2)} \leftarrow 0$
  $J^{(2,1)} \leftarrow 0$
  $J^{(2,2)} \leftarrow 0$
  for all r **do**
    $J^{(1,1)} \mathrel{+}= C^{(1,1)}_{q,r} * c_r$
    $J^{(1,2)} \mathrel{+}= C^{(1,2)}_{q,r} * c_r$
    $J^{(2,1)} \mathrel{+}= C^{(2,1)}_{q,r} * c_r$
    $J^{(2,2)} \mathrel{+}= C^{(2,2)}_{q,r} * c_r$
  $d = J^{(1,1)} * J^{(2,2)} - J^{(1,2)} * J^{(2,1)}$
  $K^{(1,1)} = J^{(2,2)} \div d$
  $K^{(1,2)} = -J^{(1,2)} \div d$
  $K^{(2,1)} = -J^{(2,1)} \div d$
  $K^{(2,2)} = J^{(1,1)} \div d$
  for all k **do**
    $t_1[k] = K_{11} E^{(1)}_{q,k} + K_{21} E^{(2)}_{q,k}$
    $t_2[k] = K_{12} E^{(1)}_{q,k} + K_{22} E^{(2)}_{q,k}$
  $t_5 = w_q * \mathrm{abs}(d)$
  for all j **do**
    $t_3 = K_{11} E^{(1)}_{q,j} + K_{21} E^{(2)}_{q,j}$
    $t_4 = K_{12} E^{(1)}_{q,j} + K_{22} E^{(2)}_{q,j}$
    for all k **do**
      $A[j,k] \mathrel{+}= t_5 * (t_1[k] * t_3 + t_2[k] * t_4)$

5.5. COFFEE integration. Although we did not emphasize this, TSFC does not generate C code directly, but an *abstract syntax tree* (AST) for C, so that COFFEE can further optimize the kernel. Since this is just a different representation of C code, the discussions in section 4 are still completely valid. COFFEE performs both low-level optimizations, such as alignment and padding to facilitate vectorization by the host C compiler, and high-level optimizations to reduce the operation count. The latter require restructuring the loops and expressions. In essence, COFFEE seeks to determine the optimal trade-off amongst common subexpressions elimination, loop-invariant code motion and factorization. For this, it analyzes a transformation space which also includes the FFC tensor representation and many of the factorizations performed by UFLACS. For further details about COFFEE and for comparative performance evaluations, see Luporini, Ham, and Kelly [30].

6. Experimental evaluation. We experimentally evaluate and compare form compilation times with the following form compilers:

- FFC quadrature representation (supports affine cells only)
- Modified version of FFC quadrature representation, developed for Firedrake to support non-affine cells.
- UFLACS

- TSFC

We disabled optional optimizations, and also included in the form compilation times the time to compile the generated C or C++ code, using GCC 5.4.0 as C and C++ compiler at `-O2` optimization level. FFC tensor representation is not included since its limitations are well known [34] (subsection 1.4), and it fails to compile our more difficult test problems due to missing features.

Our test problems consist of the left-hand sides of the following four elliptic PDEs of increasing complexity:

Helmholtz equation.

$$a = (uv + \nabla u \cdot \nabla v)\,\mathrm{d}x \tag{6.1}$$

Linear elasticity. In elastic models the solution is typically a *displacement vector* field. Let $\mathbf{u}$ and $\mathbf{v}$ be vector-valued trial and test functions, and let ε denote the symmetric part of the Jacobian of a vector field. Then the bilinear form is defined as follows:

$$\varepsilon(\mathbf{u}) = \frac{1}{2}\left[\nabla\mathbf{u} + (\nabla\mathbf{u})^T\right] \tag{6.2}$$

$$a = \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v})\,\mathrm{d}x \tag{6.3}$$

Hyperelasticity. A simple hyperelastic model. First, we define the strain energy function over the displacement vector field $\mathbf{u}$:

$$\mathbf{F} = \mathbf{I} + \nabla\mathbf{u} \qquad \triangleright \text{ Deformation gradient} \tag{6.4}$$

$$\mathbf{C} = \mathbf{F}^T\mathbf{F} \qquad \triangleright \text{ Right Cauchy-Green tensor} \tag{6.5}$$

$$\mathbf{E} = (\mathbf{C} - \mathbf{I})/2 \qquad \triangleright \text{ Euler-Lagrange strain tensor} \tag{6.6}$$

$$\Psi = \frac{\lambda}{2}[\mathrm{tr}(\mathbf{E})]^2 + \mu\,\mathrm{tr}(\mathbf{E}^2) \qquad \triangleright \text{ Strain energy function} \tag{6.7}$$

where λ and μ are the *Lamé parameters*, and $\mathbf{I}$ is the identity matrix. Now, we define the Piola-Kirchhoff stress tensors:

$$\mathbf{S} = \frac{\partial\Psi}{\partial\mathbf{E}} \qquad \triangleright \text{ Second Piola-Kirchhoff stress tensor} \tag{6.8}$$

$$\mathbf{P} = \mathbf{F}\mathbf{S} \qquad \triangleright \text{ First Piola-Kirchhoff stress tensor} \tag{6.9}$$

One could, of course, derive that $\mathbf{S} = \lambda\,\mathrm{tr}(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}$, but we leave the symbolic differentiation for UFL. Finally, the residual form of this nonlinear problem is

$$r = \mathbf{P} : \nabla\mathbf{v} - \mathbf{b} \cdot \mathbf{v} \tag{6.10}$$

where $\mathbf{b}$ is the external forcing. To assemble a left-hand side, one must linearize the residual around an *approximate* solution $\mathbf{u}$:

$$a = \delta r(\mathbf{u}; \delta\mathbf{u}) = \lim_{\epsilon\to 0} \frac{r(\mathbf{u} + \epsilon\delta\mathbf{u}) - r(\mathbf{u})}{\epsilon} \tag{6.11}$$

This bilinear form has trial function $\delta\mathbf{u}$, test function $\mathbf{v}$, and $\mathbf{u}$ is a coefficient of the form.

Holzapfel-Ogden model. A more complicated hyperelastic model described in [6]. Unlike the simple hyperelastic model, this model uses a volume-preserving right Cauchy–Green strain tensor:

$$J = \det(\mathbf{F}) \tag{6.12}$$

$$\bar{\mathbf{C}} = J^{-\frac{2}{3}}\mathbf{F}^T\mathbf{F} \tag{6.13}$$

Then defines the following rotation invariant functions:

$$I_1 = \operatorname{tr}(\bar{\mathbf{C}}) \tag{6.14}$$

$$I_{4f} = \mathbf{e}_f \cdot \bar{\mathbf{C}}\mathbf{e}_f \tag{6.15}$$

$$I_{4s} = \mathbf{e}_s \cdot \bar{\mathbf{C}}\mathbf{e}_s \tag{6.16}$$

$$I_{8fs} = \mathbf{e}_s \cdot \bar{\mathbf{C}}\mathbf{e}_f \tag{6.17}$$

where $\mathbf{e}_f$ and $\mathbf{e}_s$ denote unit vectors pointing in the local myocardial fiber and sheet directions, respectively. The strain energy function is defined as

$$\begin{aligned}\Psi =& \frac{a}{2b}\left(\exp\left[b(I_1-3)\right]-1\right)\\ &+\frac{h(I_{4f})a_f}{2b_f}\left(\exp\left[b_f(I_{4f}-1)^2\right]-1\right)\\ &+\frac{h(I_{4s})a_s}{2b_s}\left(\exp\left[b_s(I_{4s}-1)^2\right]-1\right)\\ &+\frac{a_{fs}}{2b_{fs}}\left(\exp\left[bI_{8fs}^2\right]-1\right)\end{aligned} \tag{6.18}$$

where $h(x)$ is a Heaviside function with a jump at $x = 1$, and the material parameters are

$$m = (a, b, a_f, b_f, a_s, b_s, a_{fs}, b_{fs}).$$

We define the first Piola-Kirchhoff stress tensor, and the residual form without external forcing:

$$\mathbf{P} = \frac{\partial \Psi}{\partial \mathbf{F}} \qquad \triangleright \text{Find first Piola-Kirchhoff tensor} \tag{6.19}$$

$$r = \mathbf{P} : \nabla \mathbf{v} \tag{6.20}$$

Finally, (6.11) gives the bilinear form again.

Figure 10 presents violin plots of the form compilation times, each made from 18 samples. The measurements were run on an Intel® Xeon® E5–2620 (Sandy Bridge) 2.00 GHz CPU. Due to the nature of development with Firedrake or FEniCS, the form compiler is frequently invoked in an interactive manner. Consequently, form compilation shall preferably take no more time than the fraction of a second; waiting for more than a couple of seconds is undesirable in a natural workflow.

To make the measurements as fair as possible, we configured FFC representations to skip element compilation, and for the C++ compilation we only kept the relevant `tabulate_tensor` method from the generated UFC class. However, since we did not want to change the generated code, we had to include the relevant header files. FFC representations generate code that use functions from the standard C++ header

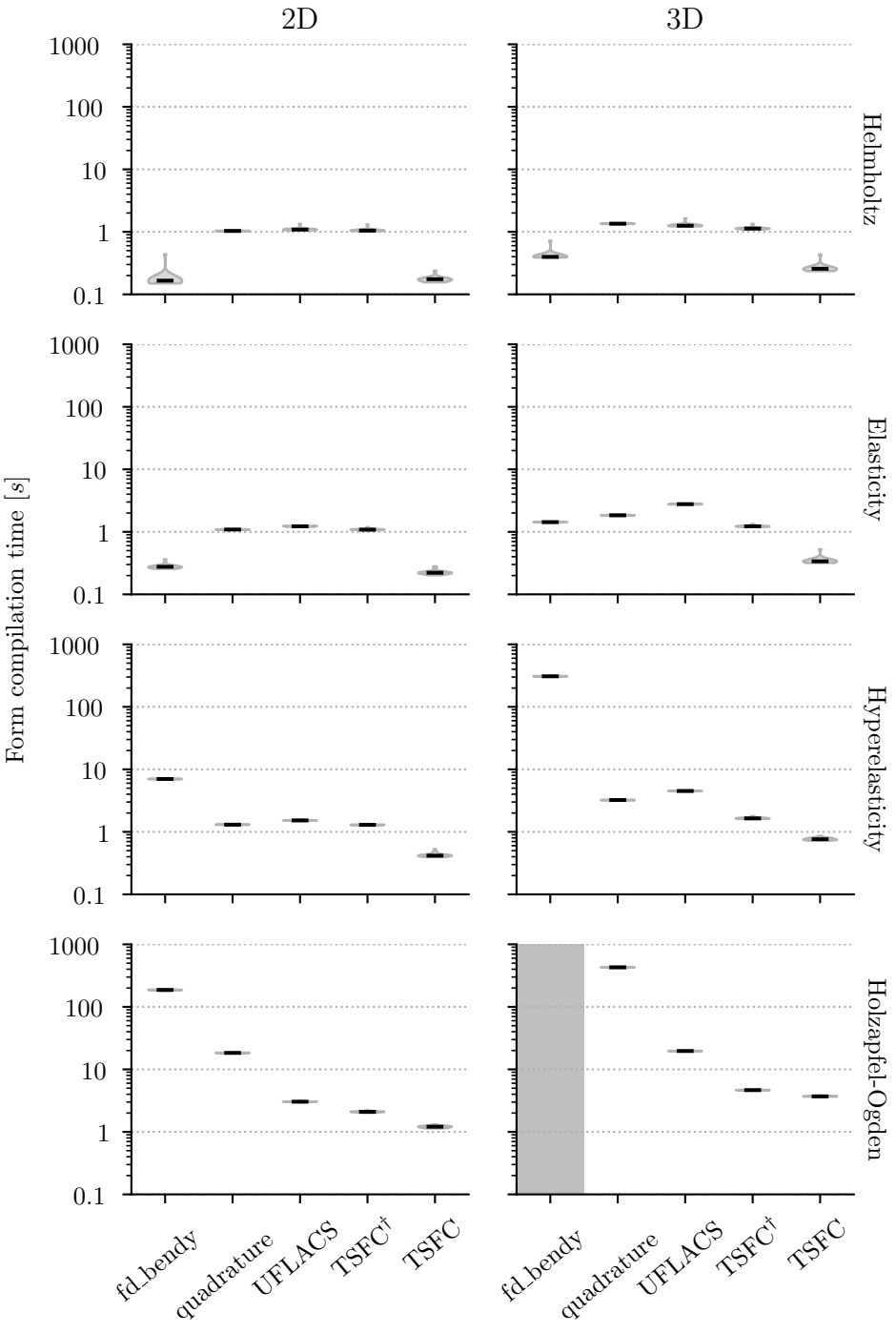

Figure 10: Form compilation time for forms of increasing complexity with the following form compilers: FFC quadrature representation, its modified version which supports non-affine cells (fd_bendy), UFLACS, TSFC, and TSFC as an FFC representation (TSFC†). 3D Holzapfel-Ogden data for fd_bendy are missing because compilation ran out of memory after more than 6 hours on a machine with 64GB of memory.

`<algorithm>`, the compilation of which takes about 0.9 seconds on the test machine. Form compilers targeting Firedrake (fd_bendy and TSFC) generate C code, and the inclusion of relevant header files induces much less overhead. This effect dominates the simplest test case, the Helmholtz equation. For further fairness, we also included TSFC as an FFC representation (shown as TSFC†), including the same set of header files as for the other FFC representations.

All form compilers compile the Helmholtz equation quickly and perform tolerably on the elastic model. The modified version of FFC quadrature with non-affine support (fd_bendy) demonstrates excessive compilation times on the simple hyperelastic model already. While non-affine support adds a huge toll to compile-time performance, even the base version of FFC quadrature becomes unmanageable as form complexity increases, as the data for Holzapfel-Ogden model demonstrate. UFLACS and TSFC, however, have tolerable compile-time performance in all cases, TSFC being somewhat faster, but typically within the same order of magnitude.

The overall compilation time of the most difficult case (Holzapfel-Ogden 3D) with TSFC can further be reduced from 3.7 to 2.2 seconds by disabling the optimization described in subsection 5.4. This configuration, however, did not yield significant benefits for the other test cases as long as C compilation times are also included.

For reproducibility, we cite archives of the exact software versions that were used to produce these results. The modified version of FFC with support for non-affine geometries requires custom branches of FFC [42] and UFL [49] which require older versions of COFFEE [39], FIAT [44], and Instant [47]. For all other form compilers and representations we used recent versions: COFFEE [40], dijitso [41], FFC [43], FIAT [45], FInAT [46], TSFC [48], and UFL [50]. The experimentation framework used to generate the violin plots is available in [15].

7. Conclusion. The experimental data demonstrate that the efficient traversal of expressions keeps the compilation time with TSFC acceptably low even for highly complicated forms. This was a crucial requirement for properly supporting non-simplicial cells as well as higher-order geometries in Firedrake.

A comprehensive analysis of run-time performance (of the generated code) is beyond the scope of this paper. For such an analysis, we refer readers to Luporini, Ham, and Kelly [30] which compares the representations of FFC, UFLACS, and an earlier and a recent version of COFFEE. TSFC is compatible with COFFEE's optimizations.

Acknowledgements. The authors would like to thank Andrew T. T. McRae for reviewing the manuscript of this paper. The integration of TSFC as an FFC representation was undertaken by Jan Blechta. TSFC relies on several recent enhancements to UFL by Martin S. Alnæs and Andrew T. T. McRae.

REFERENCES

[1] A. Allam, J. Ramanujam, G. Baumgartner, and P. Sadayappan, *Memory minimization for tensor contractions using integer linear programming*, in Proceedings 20th IEEE International Parallel Distributed Processing Symposium, April 2006, https://doi.org/10.1109/IPDPS.2006.1639717.

[2] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, *The FEniCS Project Version 1.5*, Archive of Numerical Software, 3 (2015), pp. 9–23, https://doi.org/10.11588/ans.2015.100.20553.

[3] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations*, ACM Transactions on Mathematical Software, 40 (2014), pp. 1–37, https://doi.org/10.1145/2566630, https://arxiv.org/abs/1211.4047.

[4] M. S. Alnæs and K.-A. Mardal, *On the efficiency of symbolic computations combined with code generation for finite element methods*, ACM Transactions on Mathematical Software, 37 (2010), pp. 6:1–6:26, https://doi.org/10.1145/1644001.1644007.

[5] B. Bagheri and R. Scott, *Analysa*, http://people.cs.uchicago.edu/~ridg/al/aa.html (accessed 2016-08-05).

[6] G. Balaban, M. S. Alnæs, J. Sundnes, and M. E. Rognes, *Adjoint multi-start-based estimation of cardiac hyperelastic material parameters using shear data*, Biomechanics and Modeling in Mechanobiology, 15 (2016), pp. 1509–1521, https://doi.org/10.1007/s10237-016-0780-7, https://arxiv.org/abs/1603.03796.

[7] G. Baumgartner, A. A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, *Synthesis of high-performance parallel programs for a class of* Ab Initio *quantum chemistry models*, Proceedings of the IEEE, 93 (2005), pp. 276–292, https://doi.org/10.1109/JPROC.2004.840311.

[8] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. E. Bernholdt, and R. Harrison, *Space-time trade-off optimization for a class of electronic structure calculations*, in Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, New York, NY, USA, 2002, ACM, pp. 177–186, https://doi.org/10.1145/512529.512551.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*, The MIT Press, 2001.

[10] P. Dular, C. Geuzaine, F. Henrotte, and W. Legros, *A general environment for the treatment of discrete problems and its application to the finite element method*, IEEE Transactions on Magnetics, 34 (1998), pp. 3395–3398, https://doi.org/10.1109/20.717799.

[11] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, and A. I. Krylov, *New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations*, Journal of Computational Chemistry, 34 (2013), pp. 2293–2309, https://doi.org/10.1002/jcc.23377.

[12] A. Hartono, Q. Lu, X. Gao, S. Krishnamoorthy, M. Nooijen, G. Baumgartner, D. E. Bernholdt, V. Choppella, R. M. Pitzer, J. Ramanujam, A. Rountev, and P. Sadayappan, *Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations*, in Computational Science – ICCS 2006, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds., vol. 3991 of Lecture Notes in Computer Science, Springer, 2006, pp. 267–275, https://doi.org/10.1007/11758501_39.

[13] A. Hartono, A. Sibiryakov, M. Nooijen, G. Baumgartner, D. E. Bernholdt, S. Hirata, C.-C. Lam, R. M. Pitzer, J. Ramanujam, and P. Sadayappan, *Automated Operation Minimization of Tensor Contraction Expressions in Electronic Structure Calculations*, in Computational Science – ICCS 2005, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, eds., vol. 3514 of Lecture Notes in Computer Science, Springer, May 2005, pp. 155–164, https://doi.org/10.1007/11428831_20.

[14] F. Hecht, *New development in freefem++*, Journal of Numerical Mathematics, 20 (2012), pp. 251–266, https://doi.org/10.1515/jnum-2012-0013.

[15] M. Homolya, *Experimentation framework for manuscript "TSFC: a structure-preserving form compiler," version 1*, May 2017, https://doi.org/10.5281/zenodo.573371.

[16] A. B. Kahn, *Topological sorting of large networks*, Communications of the ACM, 5 (1962), pp. 558–562, https://doi.org/10.1145/368996.369025.

[17] R. C. Kirby, *Algorithm 839: FIAT, A New Paradigm for Computing Finite Element Basis Functions*, ACM Transactions on Mathematical Software (TOMS), 30 (2004), pp. 502–516, https://doi.org/10.1145/1039813.1039820.

[18] R. C. Kirby and D. A. Ham, *FInAT Is not A Tabulator (FInAT)*, 2016, https://github.com/FInAT/FInAT.

[19] R. C. Kirby and A. Logg, *A compiler for variational forms*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 417–444, https://doi.org/10.1145/1163641.1163644, https://arxiv.org/abs/1112.0402.

[20] R. C. Kirby and A. Logg, *Efficient Compilation of a Class of Variational Forms*, ACM Transactions on Mathematical Software (TOMS), 33 (2007), https://doi.org/10.1145/1268769.1268771, https://arxiv.org/abs/1205.3014.

[21] D. E. Knuth, *The Art of Computer Programming Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 9 (1968).

[22] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, *Factor Graphs and the Sum-Product*

Algorithm, IEEE Transactions on Information Theory, 47 (2001), pp. 498–519, https://doi.org/10.1109/18.910572.

[23] C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, *Memory-Optimal Evaluation of Expression Trees Involving Large Objects*, in High Performance Computing – HiPC'99, P. Banerjee, V. K. Prasanna, and B. P. Sinha, eds., vol. 1745 of Lecture Notes in Computer Science, Springer, December 1999, pp. 103–110, https://doi.org/10.1007/978-3-540-46642-0_15.

[24] C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, *Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals*, in Languages and Compilers for Parallel Computing, L. Carter and J. Ferrante, eds., vol. 1863 of Lecture Notes in Computer Science, Springer, August 1999, pp. 350–364, https://doi.org/10.1007/3-540-44905-1_22.

[25] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan, *Memory-optimal evaluation of expression trees involving large objects*, Computer Languages, Systems and Structures, 37 (2011), pp. 63–75, https://doi.org/10.1016/j.cl.2010.09.003.

[26] C.-C. Lam, P. Sadayappan, and R. Wenger, *On Optimizing a Class of Multi-Dimensional Loops with Reduction for Parallel Execution*, Parallel Processing Letters, 07 (1997), pp. 157–168, https://doi.org/10.1142/S0129626497000176.

[27] A. Logg, K.-A. Mardal, and G. Wells, eds., *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*, vol. 84 of Lecture Notes in Computational Science and Engineering, Springer, 2012, https://doi.org/10.1007/978-3-642-23099-8.

[28] A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *FFC: the FEniCS form compiler*, in Logg et al. [27], pp. 227–238, https://doi.org/10.1007/978-3-642-23099-8.

[29] K. Long, R. Kirby, and B. van Bloemen Waanders, *Unified embedded parallel finite element computations via software-based Fréchet differentiation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3323–3351, https://doi.org/10.1137/09076920X.

[30] F. Luporini, D. A. Ham, and P. H. J. Kelly, *An algorithm for the optimization of finite element integration loops*, ACM Transactions on Mathematical Software (TOMS), 44 (2017), pp. 3:1–3:26, https://doi.org/10.1145/3054944, https://arxiv.org/abs/1604.05872.

[31] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly, *Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly*, ACM Transactions on Architecture and Code Optimization (TACO), 11 (2015), pp. 57:1–57:25, https://doi.org/10.1145/2687415, https://arxiv.org/abs/1407.0904.

[32] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris, *Generating Efficient Tensor Contractions for GPUs*, in 44th International Conference on Parallel Processing (ICPC), Sept 2015, pp. 969–978, https://doi.org/10.1109/ICPP.2015.106.

[33] K. B. Ølgaard, A. Logg, and G. N. Wells, *Automated Code Generation for Discontinuous Galerkin Methods*, SIAM Journal on Scientific Computing, 31 (2008), pp. 849–864, https://doi.org/10.1137/070710032.

[34] K. B. Ølgaard and G. N. Wells, *Optimisations for quadrature representations of finite element tensors through automated code generation*, ACM Transactions on Mathematical Software (TOMS), 37 (2010), pp. 8:1–8:23, https://doi.org/10.1145/1644001.1644009, https://arxiv.org/abs/1104.0199.

[35] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: automating the finite element method by composing abstractions*, ACM Transactions on Mathematical Software (TOMS), 43 (2016), pp. 24:1–24:27, https://doi.org/10.1145/2998441, https://arxiv.org/abs/1501.01809.

[36] B. A. Sanders, R. Bartlett, E. Deumens, V. Lotrich, and M. Ponton, *A Block-Oriented Language and Runtime System for Tensor Algebra with Very Large Arrays*, in 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2010, pp. 1–11, https://doi.org/10.1109/SC.2010.3.

[37] R. Sethi and J. D. Ullman, *The Generation of Optimal Code for Arithmetic Expressions*, Journal of the Association for Computing Machinery (JACM), 17 (1970), pp. 715–728, https://doi.org/10.1145/321607.321620.

[38] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, *Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions*, in 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), May 2013, pp. 813–824, https://doi.org/10.1109/IPDPS.2013.112.

[39] Zenodo/COFFEE, *COFFEE: a COmpiler for Fast Expression Evaluation*, Feb. 2016, https://doi.org/10.5281/zenodo.573235.

[40] Zenodo/COFFEE, *COFFEE: a COmpiler for Fast Expression Evaluation*, May 2017, https://doi.org/10.5281/zenodo.573267.
[41] Zenodo/dijitso, *A Python module for distributed just-in-time shared library building*, May 2017, https://doi.org/10.5281/zenodo.573287.
[42] Zenodo/FFC, *FFC: The FEniCS Form Compiler*, Apr. 2015, https://doi.org/10.5281/zenodo.573237.
[43] Zenodo/FFC, *FFC: The FEniCS Form Compiler*, May 2017, https://doi.org/10.5281/zenodo.573270.
[44] Zenodo/FIAT, *FIAT: FInite Element Automated Tabulator*, Feb. 2016, https://doi.org/10.5281/zenodo.573238.
[45] Zenodo/FIAT, *FIAT: FInite Element Automated Tabulator*, May 2017, https://doi.org/10.5281/zenodo.573269.
[46] Zenodo/FInAT, *FInAT: a smarter library of finite elements*, May 2017, https://doi.org/10.5281/zenodo.573266.
[47] Zenodo/Instant, *Instant is a Python module that allows for instant inlining of C and C++ code in Python*, May 2017, https://doi.org/10.5281/zenodo.573255.
[48] Zenodo/TSFC, *TSFC: The Two-Stage Form Compiler*, May 2017, https://doi.org/10.5281/zenodo.573271.
[49] Zenodo/UFL, *UFL: Unified Form Language*, Apr. 2015, https://doi.org/10.5281/zenodo.573236.
[50] Zenodo/UFL, *UFL: Unified Form Language*, May 2017, https://doi.org/10.5281/zenodo.573268.