Meta-programming for cross-domain tensor optimizations

Adilla Susungi, Norman A Rink, Albert Cohen, Jeronimo Castrillon, Claude Tadonki

Meta-programming for Cross-Domain Tensor Optimizations

Adilla Susungi
MINES ParisTech
PSL Research University
France

Norman A. Rink
Chair for Compiler Construction
Technische Universität Dresden
Germany

Albert Cohen
INRIA & ENS DI
France

Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Germany

Claude Tadonki
MINES ParisTech
PSL Research University
France

Abstract

Many modern application domains crucially rely on tensor operations. The optimization of programs that operate on tensors poses difficulties that are not adequately addressed by existing languages and tools. Frameworks such as TensorFlow offer good abstractions for tensor operations, but target a specific domain, i.e. machine learning, and their optimization strategies cannot easily be adjusted to other domains. General-purpose optimization tools such as Pluto and existing meta-languages offer more flexibility in applying optimizations but lack abstractions for tensors. This work closes the gap between domain-specific tensor languages and general-purpose optimization tools by proposing the *Tensor optimizations Meta-Language* (TeML). TeML offers high-level abstractions for both tensor operations and loop transformations, and enables flexible composition of transformations into effective optimization paths. This compositionality is built into TeML's design, as our formal language specification will reveal. We also show that TeML can express tensor computations as comfortably as TensorFlow and that it can reproduce Pluto's optimization paths. Thus, optimized programs generated by TeML execute at least as fast as the corresponding Pluto programs. In addition, TeML enables optimization paths that often allow outperforming Pluto.

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278131

CCS Concepts • **Software and its engineering → Semantics**; **Source code generation**; *Domain specific languages*; *General programming languages*;

Keywords tensor algebra, meta-programming, code generation and optimization, denotational semantics

ACM Reference Format:
Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2018. Meta-programming for Cross-Domain Tensor Optimizations. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3278122.3278131

1 Introduction

Tensors are the central data structure in many modern application domains, including scientific computing, image processing, and machine learning. Operating on tensors requires multi-dimensional and computationally intense loop nests that are difficult to optimize. Optimization strategies are complex, including not only loop transformations but also data layout and mapping decisions as well as algebraic transformations. Moreover, optimization choices frequently depend on the target platform and the application domain.

Existing frameworks such as Theano [7] or TensorFlow [2] support the development of machine learning programs that operate on tensors, but their built-in optimization strategies cannot be expected to generalize well to other applications domains. This is also the case for TACO [18], which targets applications that mix dense and sparse tensors. The state-of-the-art polyhedral loop optimizer Pluto [8] provides more flexibility in optimizing general loop nests from any application domain. However, since Pluto is a general-purpose tool, it has no abstractions for tensors, and hence suitable algebraic and layout transformations are out of its scope. Moreover, Pluto's built-in heuristics for loop optimizations limit its flexibility in composing finely tuned sequences of transformations.

High-level meta-programming languages and optimization APIs [4, 9, 10, 12, 13, 19, 21, 23, 39] offer more flexibility in expressing typical loop transformations, e.g. fusion, tiling, or interchange. Moreover, the level of abstraction for loop transformations both improves productivity in manual code optimization and eases the automation of searches for optimizations. However, existing languages and APIs often rely on an imperative programming style with limited flexibility in composing loop transformations. This makes it difficult to meet the needs of a wide range of tensor-based applications from different domains.

To address the lack of cross-domain solutions for optimizing programs that operate on tensors, this paper presents TeML, the *Tensor optimizations Meta-Language*. TeML makes tensor expressions and loop transformations first-class citizens, and it overcomes the limitations of existing solutions by embracing two key ideas. First, TeML provides high-level abstractions for characteristic tensor operations such as *contraction*, the *outer product* or the *Hadamard product* (i.e. entrywise multiplication). These abstractions encode algebraic information that is required to perform algebraic transformations. Second, TeML unifies the stages of program construction and transformation, both of which are specified in a functional programming style, thus rendering TeML's language constructs fully and flexibly composible.

After giving an overview of TeML (Section 2), we present a formal specification of TeML's semantics (Section 3), which enables clean reasoning about interactions between tensor expressions and loop transformations, and also highlights some subtleties in defining semantics for meta-languages. The power of TeML's functional style is brought to bear when we show how TeML's more abstract tensor operations and loop transformations can be defined in terms of more fundamental TeML constructs (Section 4). Using a set of characteristic benchmarks from a range of application domains, we demonstrate that TeML is as expressive as high-level tensor languages and that it offers the flexibility to compose loop optimizations such that better program performance can be achieved than with state-of-the-art tools (Section 5).

2 Overview

As a meta-programming language, TeML is designed to specify how programs in a target language (typically C) are to be generated and transformed. In particular, TeML specifies target language programs that operate on tensors; and such programs are generally formed of tensor expressions and loop nests. TeML is purposefully low-level, including no abstractions for more advanced mathematical structures that occur in the context of tensor algebra (e.g. co-/contravariant indices). Thus, TeML meta-programs operate only on two kinds of objects: *tensor expressions* and *loop nests*. As a consequence, TeML can be used as an intermediate language for a wide range of tensor algebra frameworks that implement more abstract structures and rewrite rules at a higher level.

According to the TeML grammar in Figure 1, a TeML meta-program is a sequence of statements. Statements include assignments of the form *id* **=** *expression*, and an expression produces either a tensor (*Texpression*) or a loop (*Lexpression*). Identifiers (*id*) can refer to either of these objects.

A new tensor (scalar) is introduced by **tensor** (**scalar**). The dimensions of the new tensor are specified as a list of integers, and we subsequently refer to this list as the *shape* of the tensor. All scalars and elements of tensors are assumed to be of a fixed type that can be set globally for the TeML system, e.g. to a floating-point or integer type. Tensors (and scalars) are the fundamental building blocks of complex tensor expressions, which are formed with the TeML built-ins **eq**, **vop**, and **op**. Since only identifiers are allowed as arguments, TeML specifies complex tensor expressions in a 3-address format, e.g. `t2` **=** **vop**(`t0`, `t1`, [,]).

The **vop** built-in implements arithmetic expressions and can be either of **vadd**, **vsub**, **vmul**, or **vdiv**. The optional arguments to **vop** are lists of iterators (*iters*) that are used to index tensors. For example, if `t0`, `t1` refer to tensors of shapes `[N1, N2]` and `[N3]` respectively, then **vadd**(`t0`, `t1`, [[`i1`, `i2`], [`i3`]]) represents the expression `t0[i1][i2] + t1[i3]` in the target language. The **v** in **vop** is for *virtual*, indicating that the elements of the resulting tensor expression are not stored in memory, as opposed to a *real* tensor that is backed by memory. In the previous example, the values of `t0[i1][i2] + t1[i3]`, for varying values of `i1`, `i2`, `i3`, do not reside in memory but must be computed from the elements of the real tensors `t0` and `t1`.

The **eq** built-in yields an assignment in the target language, e.g., the TeML assignment `t2` **=** **eq**(`t0`, [`i1`, `i2`] → [`i2`, `i1`]) represents the target language assignment `t2[i2][i1] = t0[i1][i2]`. As a consequence, `t2` must refer to a real tensor that has previously been introduced by `t2`**=tensor**([`N2`, `N1`]). The **op** built-in is syntactic sugar for the combination of **vop** and **eq**, i.e. **op** assigns the value of a tensor expression (viz. a virtual tensor) to a real tensor.

The identifier id_1 in the TeML assignment id_1 **=** *expression* has two meanings. First, from TeML's meta-programming perspective, id_1 is used to reference a tensor expression. Second, in the generated target program, id_1 is the name of the tensor variable that is assigned to. This default behavior can be overridden by using an assignment of the form id_1 **=** **@**id_2 : *expression*. Then, id_1 still refers to the tensor expression in TeML, but id_2 is used in the generated target program. It is not uncommon in meta-programming that names and objects acquire multiple meanings that depend on the abstraction level under consideration.

Low-level arithmetic operations such as **add**, **sub**, and **mul** give TeML a high degree of expressiveness. However, to leverage the full potential of meta-programming, it is desirable to manipulate higher-level operations, e.g. tensor

⟨*program*⟩ ::= ⟨*stmt*⟩ ⟨*program*⟩
| ϵ

⟨*stmt*⟩ ::= ⟨*id*⟩ = ⟨*expression*⟩
| ⟨*id*⟩ = @⟨*id*⟩ : ⟨*expression*⟩
| **codegen** (⟨*ids*⟩)
| **init** (...)

⟨*expression*⟩ ::= ⟨*Texpression*⟩
| ⟨*Lexpression*⟩

⟨*Texpression*⟩ ::= **scalar** ()
| **tensor** ([⟨*ints*⟩])
| **eq** (⟨*id*⟩, ⟨*iters*⟩? → ⟨*iters*⟩)
| **vop** (⟨*id*⟩, ⟨*id*⟩, [⟨*iters*⟩?, ⟨*iters*⟩?])
| **op** (⟨*id*⟩, ⟨*id*⟩, [⟨*iters*⟩?, ⟨*iters*⟩?] → ⟨*iters*⟩)

⟨*Lexpression*⟩ ::= **build** (⟨*id*⟩)
| **stripmine** (⟨*id*⟩, ⟨*int*⟩, ⟨*int*⟩)
| **interchange** (⟨*id*⟩, ⟨*int*⟩, ⟨*int*⟩)
| **fuse_outer** (⟨*id*⟩, ⟨*id*⟩, ⟨*int*⟩)
| **fuse_inner** (⟨*id*⟩, ⟨*int*⟩)
| **unroll** (⟨*id*⟩, ⟨*int*⟩)

⟨*iters*⟩ ::= [⟨*ids*⟩]

⟨*ids*⟩ ::= ⟨*id*⟩ (, ⟨*id*⟩)*

⟨*ints*⟩ ::= ⟨*int*⟩ (, ⟨*int*⟩)*

Figure 1. TeML core grammar

contraction, that abstract common tensor kernels such as the matrix product, the matricized tensor times Khatri-Rao product, or the sampled dense-dense matrix product. Contraction is also a specific example of a class of higher-level operations known as *reductions*. We defer the introduction of various high-level operations to Section 4 to avoid blowing the formal specification of the core of TeML (in Section 3) out of proportion. It is one of the strengths of TeML that complex operations can be defined compositionally from more fundamental constructs. Note, however, that virtual reduction operations, e.g. **vcontract** for contraction, must be considered fundamental, as is explained in Section 4.2.3. An identifier id_t that refers to a tensor expression is wrapped in a loop nest by id_l = **build**(id_t). The identifier id_l then refers to this loop nest, whose depth equals the number of iterators that appear in the tensor expression id_t. Building the loop nest id_l automatically numbers the iterators that appear in the tensor expression id_t: iterator `ik` is introduced by the loop at nesting level `k` inside the loop nest id_l. The range of each iterator `ik` is inferred from how it is used in the tensor expression id_t.

The loop nest id_l can be transformed by **stripmine**, **interchange**, **fuse_outer**, **fuse_inner**, and **unroll**. Loop transformations are non-destructive and thus always return a new loop nest without affecting what id_l refers to. Note that **build** generates a perfect loop nest, but the transformations **stripmine**, **interchange**, **fuse_outer**, **fuse_inner**, and **unroll** may lead to non-perfectly nested loops.

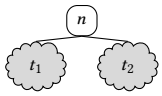


Figure 2. $\langle n, [t_1, t_2]\rangle$

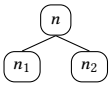


Figure 3. $\langle n, [\langle n_1, []\rangle, \langle n_2, []\rangle]\rangle$

TeML's purpose is the specification and manipulation of programs that operate on tensors, both of which are fully decoupled from the data held by tensors. Nonetheless, to initialize individual elements of tensors, TeML provides the special **init** statement, details of which we omit since they are not central to the presentation of TeML. The **codegen** statement generates the target language code for the loops that are referenced by the arguments (*ids*) of **codegen**. Presently, code generation straightforwardly translates a TeML loop into its equivalent in C, similar to the process described in [26].

3 Formal Specification

This section gives the denotational semantics of TeML. Since TeML manipulates tensor expressions and loop nests, both of which can be represented as trees, domains of trees feature prominently in TeML's semantics.

3.1 Domains of Trees

Trees are pairs $\langle n, ts\rangle$, where n is the root node and ts is a list of subtrees whose roots are the children of n. Thus, the pair $\langle n, []\rangle$, where [] is the empty list, is a leaf node. Figures 2 and 3 illustrate this notation of trees.

Tensor expressions are represented as trees whose nodes are triples (op, S, I), where op is an operation or identifier, S is the shape of the tensor expression (i.e. a list of its dimensions), and I is a list of iterators. Figures 4–6 give examples of such trees. Note that only identifiers, i.e. nodes (op, S, I) with $op = id$, may appear as leaf nodes. Internal nodes represent operations. It is not meaningful to give an iterator list for an operation, which is indicated by the special value $\bullet$ in Figures 4–6. Thus, the domain $\mathbf{T}$ of tensor expressions can be given the following recursive definition:

$$\mathbf{T} = \{\, \langle (op, S, I), ts\rangle \mid (ts = []) \vee (ts = [t_1, \ldots, t_k] \wedge t_i \in \mathbf{T}),\ I \neq \bullet \Rightarrow ts = [] \wedge op = id \,\}. \quad (1)$$

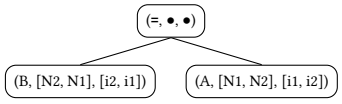
For identifiers we also allow $I = \epsilon$, indicating that the iterator list has not been set yet. Note that $\epsilon \neq []$, and also $\epsilon \neq \bullet$.

Shapes can also take the special value $\bullet$. This happens only for tree nodes (=, $\bullet$, $\bullet$) that represent assignment operations. TeML assignments do not produce tensor-valued expressions in the target language and thus have no meaningful shape (viz. dimensions) at the meta-level.

Loop nests are also represented as trees of a certain structure: the nodes are iterators and the children of a node are either loop nests or tensor expressions. Thus, the domain $\mathbf{L}'$ of loop nests can be recursively defined as

$$\mathbf{L}' = \{\, \langle id, [x_1, \ldots, x_k]\rangle \mid x_i \in \mathbf{L}' \cup \mathbf{T} \,\}. \quad (2)$$

```
A = tensor([N1, N2])
B = eq(A, [i1, i2] -> [i2, i1])
```



$\mathcal{E}_l[\![\textbf{build}(B)]\!]\sigma_2 = \langle$i1, $[\langle$i2, $[\sigma_2(B)]\rangle]\rangle$:

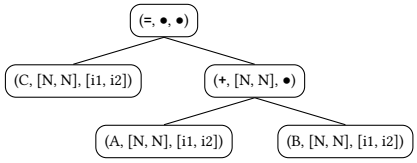
```
for (int i1 = 0; i1 <= (N1-1); i1++)
  for (int i2 = 0; i2 <= (N2-1); i2++)
    B[i2][i1] = A[i1][i2];
```

$\sigma_1 = \mathcal{P}_{stmt}[\![A = \textbf{tensor}([N1, N2])]\!]\emptyset = \{ A \mapsto \langle (A, [N1, N2], \epsilon), [] \rangle \}$

$\sigma_2 = \mathcal{P}_{stmt}[\![B = \textbf{eq}(A, [i1, i2] \to [i2, i1])]\!]\sigma_1$
$= \{ A \mapsto \langle (A, [N1, N2], \epsilon), [] \rangle,$
$B \mapsto \langle (=, \bullet, \bullet), [\langle (B, [N2, N1], [i2, i1]), [] \rangle,$
$\langle (A, [N1, N2], [i1, i2]), [] \rangle] \rangle \}$

Figure 4. Matrix transposition implemented with **eq**. The tree on the left depicts $\sigma_2(B)$. The target language (C) code at the bottom results from **build**(B).

```
A = tensor([N, N])
B = tensor([N, N])
C = tensor([N, N])
D = @C:add(A, B, [[i1, i2], [i1, i2]] -> [i1, i2])
```



$\mathcal{E}_l[\![\textbf{build}(D)]\!]\sigma_4 = \langle$i1, $[\langle$i2, $[\sigma_4(D)]\rangle]\rangle$:

```
for (int i1 = 0; i1 <= (N-1); i1++)
  for (int i2 = 0; i2 <= (N-1); i2++)
    C[i2][i1] = A[i1][i2] + B[i1][i2];
```

$\sigma_1 = \mathcal{P}_{stmt}[\![A = \textbf{tensor}([N, N])]\!]\emptyset = \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle \}$

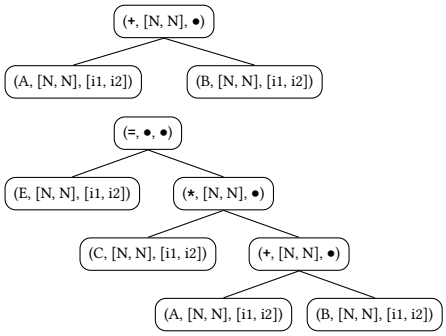
$\sigma_2 = \mathcal{P}_{stmt}[\![B = \textbf{tensor}([N, N])]\!]\sigma_1$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle \}$

$\sigma_3 = \mathcal{P}_{stmt}[\![C = \textbf{tensor}([N, N])]\!]\sigma_2$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle,$
$C \mapsto \langle (C, [N, N], \epsilon), [] \rangle \}$

$\sigma_4 = \mathcal{P}_{stmt}[\![D = \texttt{@}C : \textbf{add}(A, B, [[i1, i2], [i1, i2]] \to [i1, i2])]\!]\sigma_3$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle,$
$C \mapsto \langle (C, [N, N], \epsilon), [] \rangle,$
$D \mapsto \langle (=), \bullet, \bullet), [\langle (C, [N, N], [i1, i2]), [] \rangle, y] \rangle \},$
where $y = \langle (+, [N, N], \bullet), [\langle (A, [N, N], [i1, i2]), [] \rangle,$
$\langle (B, [N, N], [i1, i2]), [] \rangle] \rangle$

Figure 5. The `@`*id* construct. The tree on the left depicts $\sigma_4(D)$. The C code on the bottom left results from **build**(D).

```
A = tensor([N, N])
B = tensor([N, N])
C = tensor([N, N])
D = vadd(A, B, [[i1, i2], [i1, i2]])
E = mul(C, D, [[i1, i2], ] -> [i1, i2])
```



$\mathcal{E}_l[\![\textbf{build}(E)]\!]\sigma_5 = \langle$i1, $[\langle$i2, $[\sigma_5(E)]\rangle]\rangle$:

```
for (int i1 = 0; i1 <= (N-1); i1++)
  for (int i2 = 0; i2 <= (N-1); i2++)
    E[i1][i2] = C[i1][i2] * (A[i1][i2] + B[i1][i2]);
```

$\sigma_1 = \mathcal{P}_{stmt}[\![A = \textbf{tensor}([N, N])]\!]\emptyset = \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle \}$

$\sigma_2 = \mathcal{P}_{stmt}[\![B = \textbf{tensor}([N, N])]\!]\sigma_1$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle \}$

$\sigma_3 = \mathcal{P}_{stmt}[\![C = \textbf{tensor}([N, N])]\!]\sigma_2$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle,$
$C \mapsto \langle (C, [N, N], \epsilon), [] \rangle \}$

$\sigma_4 = \mathcal{P}_{stmt}[\![D = \textbf{vadd}(A, B, [[i1, i2], [i1, i2]])]\!]\sigma_3$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle,$
$C \mapsto \langle (C, [N, N], \epsilon), [] \rangle,$
$D \mapsto \langle (+, [N, N], \bullet), [\langle (A, [N, N], [i1, i2]), [] \rangle,$
$\langle (B, [N, N], [i1, i2]), [] \rangle] \rangle \}$

$\sigma_5 = \mathcal{P}_{stmt}[\![E = \textbf{mul}(C, D, [[i1, i2],] \to [i1, i2])]\!]\sigma_4$
$= \{ A \mapsto \langle (A, [N, N], \epsilon), [] \rangle,\ B \mapsto \langle (B, [N, N], \epsilon), [] \rangle,$
$C \mapsto \langle (C, [N, N], \epsilon), [] \rangle,$
$D \mapsto \langle (+, [N, N], \bullet), [\langle (A, [N, N], [i1, i2]), [] \rangle,$
$\langle (B, [N, N], [i1, i2]), [] \rangle] \rangle,$
$E \mapsto \langle (=, \bullet, \bullet), [\langle (E, [N, N], [i1, i2]), [] \rangle,$
$\langle (*, [N, N], \bullet), [\langle (C, [N, N], [i1, i2]), [] \rangle, \sigma_4(D)] \rangle] \rangle \}$

Figure 6. More complex tensor expression, including assignment to tensor E. The trees on the left depict $\sigma_4(D)$ (top tree) and $\sigma_5(E)$ (bottom tree) respectively. The target language (C) code on the bottom left results from **build**(E).

Figures 4–6 also give the tree representations (underlined, in the $\langle\,,[\,]\rangle$ notation) of the perfect loop nests returned by **build**. Non-perfect loop nests may result after applying loop transformations. While $\mathbf{L}'$ includes non-perfect loop nests, it does not include assignment operations at the top level of a TeML program, i.e. assignment operations not nested inside any loop. To capture such corner cases, we take as the domain of loops

$$\mathbf{L} = \mathbf{L}' \cup \{\, \langle \bullet, [x_1, \ldots, x_k]\rangle \mid x_i \in \mathbf{L}' \cup \mathbf{T} \,\}\,. \tag{3}$$

The trees $\langle \bullet, [x_1, \ldots, x_k]\rangle$ represent sequences of top-level loops and tensor expressions $x_1, \ldots, x_k$, i.e. loops and tensor expressions not nested inside a loop.

So far we have always referred to iterators by their names, and TeML's naming convention for iterators was introduced in Section 2. However, the iterators that appear as nodes in the tree representation of loop nests carry additional information, namely their inferred ranges. The range of an iterator is a triple (lb, ub, st) consisting of integer values lb (lower bound), ub (upper bound), and st (step).

3.2 State

The state of a TeML meta-program maps identifiers to trees representing either tensor expressions or loop nests. Thus, the domain S of states is defined as

$$\mathbf{S} \;=\; \mathit{identifier} \to (\mathbf{T} + \mathbf{L})\,. \tag{4}$$

Hence, a specific state $\sigma \in \mathbf{S}$ is formally a function

$$\sigma : \mathit{identifier} \to (\mathbf{T} + \mathbf{L})\,. \tag{5}$$

3.3 Valuation Functions

We now specify the behavior of TeML programs in terms of valuation functions. For each syntactic category (statement, expression etc.) there is a valuation function that defines how a syntactic entity manipulates a specific state $\sigma \in \mathbf{S}$:

$$\mathcal{P}_{prog} : \mathit{program} \to (\mathbf{S} \to \mathbf{S})\,, \tag{6}$$
$$\mathcal{P}_{stmt} : \mathit{stmt} \to (\mathbf{S} \to \mathbf{S})\,, \tag{7}$$
$$\mathcal{E}_t : \mathit{Texpression} \to (\mathbf{S} \to \mathbf{T})\,, \tag{8}$$
$$\mathcal{E}_l : \mathit{Lexpression} \to (\mathbf{S} \to \mathbf{L})\,. \tag{9}$$

$\mathcal{P}_{prog}$ is given in Figure 7 and is straightforward. The definition of $\mathcal{P}_{stmt}$ is split across Figures 8 and 9 since it depends on whether the right-hand side of an assignment is a tensor or a loop expression.

$$\mathcal{P}_{prog}[\![\epsilon]\!] = \mathrm{id}_{\mathbf{S}} \tag{10}$$
$$\mathcal{P}_{prog}[\![s\; p]\!] = \mathcal{P}_{prog}[\![p]\!] \circ \mathcal{P}_{stmt}[\![s]\!] \tag{11}$$

Figure 7. Definition of $\mathcal{P}_{prog}$. ϵ denotes an empty program, s a statement, and p a program; $\circ$ is function composition.

$$\mathcal{E}_t[\![\texttt{scalar}()]\!] = \lambda\sigma.\langle(\square, [], \bullet), []\rangle \tag{12}$$

$$\mathcal{E}_t[\![\texttt{tensor}(S)]\!] = \lambda\sigma.\langle(\square, S, \epsilon), []\rangle \tag{13}$$

$$\begin{aligned}\mathcal{E}_t[\![\texttt{eq}(t, I_0 \to I_1)]\!] = {}& \\ \lambda\sigma.\mathrm{let}\ & \langle(op, S, I'), ys\rangle = \sigma(t) \\ & y = \langle(op, S, I''), ys\rangle \\ & x = \langle(\square, S', I_1), []\rangle \\ \mathrm{in}\ & \langle(=, \bullet, \bullet), [x, y]\rangle\,, \\ \mathrm{where}\ & \begin{cases} I' \neq \epsilon \wedge I'' = I'\,, & \text{if } I_0 = \epsilon \\ I' = \epsilon \wedge I'' = I_0\,, & \text{if } I_0 \neq \epsilon \end{cases}\end{aligned} \tag{14}$$

$$\begin{aligned}\mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I_0, I_1])]\!] = {}& \\ \lambda\sigma.\mathrm{let}\ & \langle(op_0, S_0, I_0'), ys_0\rangle = \sigma(t_0) \\ & \langle(op_1, S_1, I_1'), ys_1\rangle = \sigma(t_1) \\ & y_0 = \langle(op_0, S_0, I_0''), ys_0\rangle \\ & y_1 = \langle(op_1, S_1, I_1''), ys_1\rangle \\ \mathrm{in}\ & \langle(\texttt{op}, S', \bullet), [y_0, y_1]\rangle\,, \\ \mathrm{where}\ & \begin{cases} I_i' \neq \epsilon \wedge I_i'' = I_i'\,, & \text{if } I_i = \epsilon \\ I_i' = \epsilon \wedge I_i'' = I_i\,, & \text{if } I_i \neq \epsilon \end{cases}\end{aligned} \tag{15}$$

$$\begin{aligned}\mathcal{E}_t[\![\texttt{op}(t_0, t_1, [I_0, I_1] \to I_2)]\!] = {}& \\ \lambda\sigma.\mathrm{let}\ & x = \mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I_0, I_1])]\!]\sigma \\ \mathrm{in}\ & \mathcal{E}_t[\![\texttt{eq}(t, \epsilon \to I_2)]\!]\,(\sigma\{t \mapsto x\})\,, \\ & \text{where } t \text{ is an identifier not in } \mathit{dom}(\sigma)\end{aligned} \tag{16}$$

$$\begin{aligned}\mathcal{P}_{stmt}[\![id = e_t]\!] = \lambda\sigma.\mathrm{let}\ & x = \mathcal{E}_t[\![e_t]\!]\sigma \\ & x' = x[id/\square] \\ \mathrm{in}\ & \sigma\{id \mapsto x'\}\end{aligned} \tag{17}$$

$$\begin{aligned}\mathcal{P}_{stmt}[\![id_1 = @id_2 : e_t]\!] = \lambda\sigma.\mathrm{let}\ & x = \mathcal{E}_t[\![e_t]\!]\sigma \\ & x' = x[id_2/\square] \\ \mathrm{in}\ & \sigma\{id_1 \mapsto x'\}\end{aligned} \tag{18}$$

Figure 8. Valuation functions $\mathcal{E}_t$ and $\mathcal{P}_{stmt}$ for tensor expressions. The placeholder $\square$ is filled with an identifier by $\mathcal{P}_{stmt}$. The shape S' in **eq** and **vop** must be inferred.

3.3.1 Tensor Expressions

Figure 8 defines the valuation function $\mathcal{E}_t$. Given a tensor expression as its argument, $\mathcal{E}_t$ constructs the tree that represents this tensor expression. The tree that is constructed for **scalar** consists of the node $(\square, [], \bullet)$, which sets the shape of a scalar to the empty list [] and forbids iterators to index scalars since the last component of the triple is $\bullet$. When a tree is constructed for **tensor**, the last component of the triple is left empty, indicated by ϵ in Equation (13). A list of iterators is filled in for ϵ when the tensor is used in an expression. The example in Figure 4 demonstrates this: the tensor referred to by A has no iterator list (in either state σ_1

or σ_2); but when A is used in the expression **eq**, an iterator list is filled in based on the arguments of **eq** (cf. state σ_2).

The symbol □ denotes a placeholder for an identifier, which can only be filled in once the identifier on the left-hand side of an assignment has been seen. Hence, $\mathcal{P}_{stmt}$ (discussed below) is responsible for filling in an identifier for □.

For **eq**, the function $\mathcal{E}_t$ constructs a tree representing an assignment operation. This also uses a placeholder to defer filling in the identifier for the tensor on the left-hand side of the constructed assignment. The grammar in Figure 1 allows the iterator list I_0 to be absent, i.e. $I_0 = \epsilon$. The where-clause after Equation (14) specifies when this is possible. If $I_0 = \epsilon$, the identifier t must refer to a tensor expression that already has a valid iterator list ($I' \neq \epsilon$), and this then becomes the iterator list of y ($I'' = I'$). If $I_0 \neq \epsilon$, then t must refer to a tensor expression with no iterator list ($I' = \epsilon$), and then I_0 is filled in as the iterator list of y ($I'' = I_0$). The optional arguments I_0 and I_1 of **vop** are handled in the same way.

As pointed out in Section 2, **op** is syntactic sugar for **vop** followed by **eq**. Equation (16) makes this precise. The notation $\sigma\{t \mapsto x\}$ means that the map σ is augmented with a mapping of the identifier t to x. Generally, an existing mapping for t in σ ($t \in dom(\sigma)$) is overwritten. In Equation (16), however, t is chosen to be a *fresh* identifier not already mapped by σ.

Note that the arguments I_0, I_1 of **op** are simply passed on to **vop** in Equation (16). The **mul** operation in Figure 6 is an example of the use of an emtpy iterator list, i.e. $I_1 = \epsilon$.

$\mathcal{P}_{stmt}$ produces a new state from its argument σ by adding a new mapping for an identifier (id in Equation (17) or id_1 in Equation (18)). The target of this new mapping is an expression tree x' that is produced by first evaluating $\mathcal{E}_t[\![e_t]\!]$ and then, in Equation (17), replacing potential occurrences of □ with the identifier id (in symbols: $x[id/□]$). Now, id occurs in the expression tree x' and will thus make it into the target language program generated by TeML; but id is also mapped in the new state $\sigma\{id \mapsto x'\}$. This is the double meaning of id (cf. Section 2) made precise. Equation (18) does not lead to this double meaning since □ is replaced with id_2 from the **@**id_2 construct, of which Figure 5 gives a worked example. Generally, the right panes of Figures 4–6 give step-by-step evaluations of $\mathcal{P}_{prog}$, broken down into evaluations of $\mathcal{P}_{stmt}$.

Since TeML assignments have a 3-address format, nested evaluations of $\mathcal{E}_t$ do not occur. Therefore, there is never more than one occurrence of □ in the tree x in Equations (17), (18).

In Equation (14) the shape S' must be inferred from the properties of y; and S' in Equation (15) must be inferred from y_0, y_1. Shape inference for tensor expressions is straightforward and follows a strategy analogous to the typing of tensor expressions in [25]. Inference fails for malformed tensor expressions, in which case TeML cannot generate a valid target language program. Also analogous to [25], the TeML assignments in Equations (17), (18) are malformed if id or id_2, respectively, has not been introduced with **tensor**.

$$\mathcal{E}_l[\![\texttt{build}(t)]\!] = \lambda\sigma.\text{let } r = \text{“number of iterators in } \sigma(t)\text{”} \\ i_k = (0, ub_k, 1) \text{ for } k = 1, \ldots, r \\ \text{in } \langle i_1, \ldots, \langle i_r, [\sigma(t)]\rangle \ldots \rangle, \\ \text{where } \sigma(t) = \langle(=, \bullet, \bullet), [x, y]\rangle \tag{19}$$

$$\mathcal{E}_l[\![\texttt{stripmine}(l, r, v)]\!] = \\ \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_r, xs\rangle \ldots \rangle = \sigma(l) \\ (b, e, 1) = i_r \\ i'_r = (0, (e-b)/v - 1, 1) \\ i'_{r+1} = (b + v \cdot i'_r, b + v \cdot i'_r + (v-1), 1) \\ \text{in } \langle i_1, \ldots \langle i'_r, [\langle i'_{r+1}, xs\rangle]\rangle \ldots \rangle \tag{20}$$

$$\mathcal{E}_l[\![\texttt{interchange}(l, r_1, r_2)]\!] = \\ \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r_1}, \ldots \langle i_{r_2}, xs\rangle \ldots \rangle \ldots \rangle = \sigma(l) \\ \text{in } \langle i_1, \ldots \langle i_{r_2}, \ldots \langle i_{r_1}, xs\rangle \ldots \rangle \ldots \rangle \tag{21}$$

$$\mathcal{E}_l[\![\texttt{fuse\_outer}(l_1, l_2, r)]\!] = \\ \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_r, xs\rangle \ldots \rangle = \sigma(l_1) \\ \langle i_1, \ldots \langle i_r, ys\rangle \ldots \rangle = \sigma(l_2) \\ \text{in } \langle i_1, \ldots \langle i_r, xs \mid\mid ys\rangle \ldots \rangle \tag{22}$$

$$\mathcal{E}_l[\![\texttt{fuse\_inner}(l, r)]\!] = \\ \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r-1}, xs\rangle \ldots \rangle = \sigma(l) \\ [\langle i_r, xs_1\rangle, \ldots, \langle i_r, xs_n\rangle] = xs \\ \text{in } \langle i_1, \ldots \langle i_{r-1}, [\langle i_r, xs_1 \mid\mid \ldots \mid\mid xs_n\rangle]\rangle \ldots \rangle \tag{23}$$

$$\mathcal{E}_l[\![\texttt{unroll}(l, r)]\!] = \\ \lambda\sigma.\text{let } \langle i_1, \ldots \langle i_{r-1}, [\langle i_r, xs\rangle]\rangle \ldots \rangle = \sigma(l) \\ (b, e, s) = i_r \\ k = (e-b)/s + 1 \\ \text{in } \langle i_1, \ldots \langle i_{r-1}, xs'_0 \mid\mid \ldots \mid\mid xs'_{k-1}\rangle \ldots \rangle, \tag{24}$$

$$\text{where } xs'_j \text{ is obtained from } xs \text{ by replacing} \\ \text{the iterator } i_r \text{ with the constant } s \cdot j + b \tag{25}$$

$$\mathcal{P}_{stmt}[\![id = e_l]\!] = \lambda\sigma.\text{let } x = \mathcal{E}_l[\![e_l]\!]\sigma \text{ in } \sigma\{id \mapsto x\} \tag{26}$$

Figure 9. Valuation functions $\mathcal{E}_l$ and $\mathcal{P}_{stmt}$ for loop expressions. The symbol || denotes concatenation of lists.

3.3.2 Loop Expressions

The definitions of $\mathcal{E}_l$ for loop expressions in Figure 9 freely identify iterators with triples (lb, ub, st) that specify iterator ranges (cf. the last paragraph of Section 3.1). When a loop nest is built around a tensor expression in Equation (19), the upper bounds ub_k are inferred based on the positions in which iterators are used to index tensors in the expression $\sigma(t)$. Note that the where-clause after Equation (19) enforces that loop nests are only built around assignments of tensors. Example evaluations of $\mathcal{E}_l[\![\texttt{build}(id)]\!]$ appear (underlined) in the bottom left corners of Figures 4–6, together with the corresponding loop nests in the target language, i.e. C.

Equation (20) implements a version of stripmining that does not transform tensor indices inside tensor expressions. The definition of **interchange** in Equation (21) is straightforward: it swaps iterators. Note that **fuse_outer** in Equation (22) fuses identical outer iterators of two loop nests, and **fuse_inner** in Equation (23) fully fuses loops at nesting level r. The tree patterns on the left-hand sides in the let-expressions in Equations (22) and (23) enforce that the loop nests $\sigma(l_1)$, $\sigma(l_2)$, and $\sigma(l)$ have specific shapes that are required for fusion to be possible.

The definition for **unroll** in Equation (25) is standard. Note that unrolling the outermost loop, i.e. $r = 1$ in Equation (25), generally results in a sequence of loops and tensor expressions that are not nested inside another loop and thus has the form $\langle \bullet, [x_1, \ldots, x_k] \rangle \in \mathbf{L}$ (cf. Equation (3)).

The definition of $\mathcal{P}_{stmt}$ in Equation (26) is completely standard. No placeholders for identifiers are required for loop expressions. This is ultimately because loops have no meaningful names at the level of target language programs.

3.3.3 Parallelization

The TeML core language (cf. Figure 1) can be extended with the loop expressions **parallelize** and **vectorize** that specify that a loop be parallelized or vectorized, respectively, using pragmas or intrinsic functions. Our representation of loops as trees in $\mathbf{L}$ does not capture this, which is not a problem since neither parallelization nor vectorization changes the structure of a loop. Further work is required to enable more fine-grained control over parallelization, including for instance the generation of atomic sections or optimized parallel reductions. This would neccesitate extending both TeML's expressivenes and its code generation process.

3.4 Type Safety in TeML

We have defined the valuation functions in a λ-calculus with a primitive tree type that has the constructor $\langle\,,[\,]\rangle$. In the let-expressions, we have made extensive use of pattern matching against this constructor. Additional constraints are enforced by where-clauses. Whenever pattern matching fails or a where-clause is not satisfied, the TeML program that is being evaluated is malformed. A TeML program is also malformed when shape inference fails or when real tensors are assigned to that have not previously been introduced with **tensor**, as discussed at the end of Section 3.3.1.

TeML detects whether a program is malformed while the program is being evaluated. This amounts to dynamic type checking, where *dynamic* means during the execution of a meta-program, i.e. before a target language program has even been generated.

4 Compositional Definitions

In this section, we extend TeML with additional loop and tensor expressions that implement more abstract operations on loops and tensors. These abstract operations enhance TeML's expressiveness and, from the perspective of a TeML implementation, are considered built into the language. Nonetheless, the key observation in this section is that more abstract operations can be defined in terms of the TeML core language by means of the composition in Equation (11). This facilitates a modular implementation of TeML and also demonstrates the flexibility of the language.

4.1 Loop Transformations

The loop transformations in the TeML core language, as defined in Figure 9, appear to be rather restrictive. For example, **interchange** only operates on immediately adjacent loops, and **unroll** performs complete unrolling of a loop. Using loop tiling as an example, we now demonstrate how common, more flexible loop transformations can be composed from the ones in the core language. In order to increase data locality, tiling organizes the iteration space of a loop nest into blocks. This can be achieved by multiple applications of stripmining, each of which introduces blocks into a single loop, cf. Equation (20). The auxiliary transformation **stripmine_n** expands into the multiple applications of stripmining required for tiling,

$$\mathcal{P}_{stmt}[\![l' = \texttt{stripmine\_n}(l, n, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_1 = \texttt{stripmine}(l, 1, v) \\ l_2 = \texttt{stripmine}(l_1, 3, v) \\ \ldots \\ l_{n-1} = \texttt{stripmine}(l_{n-2}, 2(n-1)-1, v) \\ l' = \texttt{stripmine}(l_{n-1}, 2n-1, v) \end{array} \right]\!\!\right]. \quad (27)$$

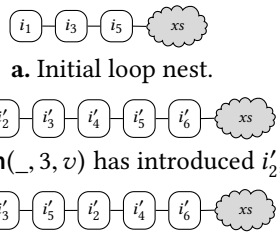
Note that if d is the depth of the original (perfect) loop nest l, then the depth of the resulting loop nest l' is $d + n$.

To arrange the new loops that stripmining has introduced into the right order for tiling, several interchanges are needed. To this end, we introduce **interchange_n** that permutes an iterator i in a loop nest through the next n iterators (towards the innermost loop),

$$\mathcal{P}_{stmt}[\![l' = \texttt{interchange\_n}(l, r, n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_1 = \texttt{interchange}(l, r, r+1) \\ l_2 = \texttt{interchange}(l_1, r+1, r+2) \\ \ldots \\ l_{n-1} = \texttt{interchange}(l_{n-2}, r+n-2, \\ \qquad\qquad r+n-1) \\ l' = \texttt{interchange}(l_{n-1}, r+n-1, \\ \qquad\qquad r+n) \end{array} \right]\!\!\right]. \quad (28)$$

Finally, TeML's loop transformation **tile** can be defined,

$$\mathcal{P}_{stmt}[\![l' = \texttt{tile}(l, v)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} l_0 = \texttt{stripmine\_n}(l, d, v) \\ l_1 = \texttt{interchange\_n}(l_0, 2, 2d-2) \\ l_2 = \texttt{interchange\_n}(l_1, 3, 2d-3) \\ \ldots \\ l_{d-1} = \texttt{interchange\_n}(l_{d-2}, d, d)) \\ l' = \texttt{interchange\_n}(l_{d-1}, d+1, d-1) \end{array} \right]\!\!\right], \quad (29)$$



a. Initial loop nest.

b. **`stripmine_n`**$(\_, 3, v)$ has introduced i'_2, i'_4, and i'_6.

c. After three times **`interchange_n`**.

Figure 10. Tiling.

where d is the depth of the original loop nest l, and the resulting loop nest l' has depth $2d$. Figure 10 demonstrates how **`tile`** handles a loop nest of initial depth 3. First, stripmining introduces the additional iterators i'_2, i'_4, and i'_6 in Figure 10b. Then, three applications of **`interchange_n`** move these iterators towards the deep positions in the loop nest, cf. Figure 10c.

Other well-known loop transformations, e.g. partial unrolling and unroll-and-jam, can be composed from the transformations defined in Figure 9 in a similar fashion.

4.2 Tensor Operations

Tensor operations that are more abstract than the fundamental arithmetic operations, such as **`add`** and **`mul`**, allow algorithms to be expressed more concisely in TeML. In particular, more abstract operations largely hide the explicit, and therefore error-prone, manipulation of iterator lists. As was the case for loop transformations, abstract tensor operations can be defined in terms of TeML's core operations, requiring only a few simple side rules for iterator lists and shapes that a TeML implementation can easily check.

4.2.1 Entrywise Operations

If the tensors t_0 and t_1 have the same shapes, the core arithmetic operations **`add`**, **`sub`**, **`mul`**, and **`div`** can be applied to all entries (viz. components) of $t0$ and $t1$ simultaneously. Let r denote the length of the shapes of t_0 and t_1. The entrywise application of arithmetic operations is enforced in the following definitions by the fact that the same iterator list $I = [\mathtt{i1}, \ldots \mathtt{ir}]$ is used for both $t0$ and $t1$,

$$\mathcal{E}_t[\![\texttt{ventrywise\_op}(t_0, t_1)]\!] = \mathcal{E}_t[\![\texttt{vop}(t_0, t_1, [I, I])]\!], \quad (30)$$

$$\mathcal{E}_t[\![\texttt{entrywise\_op}(t_0, t_1)]\!] = \mathcal{E}_t[\![\texttt{op}(t_0, t_1, [I, I] \to I)]\!]. \quad (31)$$

Theses formulae and their implementations in TeML generalize to arbitrary numbers of arguments.

4.2.2 Transposition

Transposition amounts to reordering an iterator list:

$$\mathcal{E}_t[\![\texttt{transpose}(t, [[r_1, r_2], \ldots, [r_{2k-1}, r_{2k}]])]\!] = \mathcal{E}_t[\![\texttt{eq}(t, I \to I')]\!], \quad (32)$$

where I is a list of fresh iterators for t, and I' is obtained from I by swapping pairs of iterators at positions r_{2i-1} and r_{2i}, for $i = 1, \ldots, k$. Since **`transpose`** is defined in terms of **`eq`**, it results in an assignment to a real tensor in the target language program. The corresponding virtual operation **`vtranspose`** does not introduce any operations into the target language program, it simply causes TeML to reorganize iterator lists.

4.2.3 Contraction

Contraction is central to many complex algorithms that operate on tensors. In particular, the dot product and matrix multiplication are low-dimensional instances of contraction.

Let S_0 and S_1 be the shapes of tensors t_0 and t_1, and let these shapes have lengths s_0 and s_1 respectively. The tensors t_0 and t_1 can be contracted along dimensions r_0 and r_1, respectively, if the r_0-th entry of S_0 equals the r_1-th entry of S_1. Then,

$$\mathcal{P}_{stmt}[\![t' = \texttt{contract}(t_0, t_1, [r_0, r_1])]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} t_2 = \texttt{vmul}(t_0, t_1, [I, J]) \\ t' = \texttt{add}(t', t_2, [I', \epsilon] \to I') \end{array} \right]\!\!\right], \quad (33)$$

where $I = [\mathtt{i0}, \ldots, \mathtt{i}(r_0 - 1), \mathtt{k}, \mathtt{i}(r_0 + 1), \ldots, \mathtt{i}s_0]$,
$J = [\mathtt{j0}, \ldots, \mathtt{j}(r_1 - 1), \mathtt{k}, \mathtt{j}(r_1 + 1), \ldots, \mathtt{j}s_1]$,
$I' = (I \backslash \{\mathtt{k}\})\ ||\ (J \backslash \{\mathtt{k}\})$.

The iterator k appears at position r_0 in I, and at position r_1 in J; it no longer appears in the iterator list I' for the resulting tensor t'. Contraction of more than one pair of dimensions is defined analogously, but the formulae become unwieldy.

Contraction is an example of a reduction operation and therefore crucially requires an accumulator. In Equation (33), the resulting tensor t' also acts as the accumulator, and thus t' appears both as the result and the argument of the **`add`** operation. Because of the **`add`** in Equation (33), t' is a real tensor, which in the target language program is backed by memory. This explains why the virtual counterpart **`vcontract`** cannot be given a definition in terms of more fundamental TeML operations: in a virtual operation, TeML does not have at its disposal a real tensor that can play the role of the accumulator. Hence, **`vcontract`** is considered fundamental in TeML, as already pointed out in Section 2.

Analogous statements hold for other reduction operations that TeML can be extended with by suitable compositional definitions.

4.2.4 Outer Product

The outer product, often simply referred to as the *tensor product*, combines the components of tensors $t_1, \ldots, t_n$ into a single tensor whose shape then is the concatenation of the shapes of $t_1, \ldots, t_n$. In formulae,

$$\mathcal{P}_{stmt}[\![t' = \texttt{vouterproduct}(t_1, \ldots, t_n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} s_{n-1} = \texttt{vmul}(t_{n-1}, t_n, [I_{n-1}, I_n]) \\ s_{n-2} = \texttt{vmul}(t_{n-2}, s_{n-1}, [I_{n-2}, \epsilon]) \\ \ldots \\ s_2 = \texttt{vmul}(t_2, s_3, [I_2, \epsilon]) \\ t' = \texttt{vmul}(t_1, s_2, [I_1, \epsilon] \to I') \end{array} \right]\!\!\right], \quad (34)$$

$$\mathcal{P}_{stmt}[\![t' = \texttt{outerproduct}(t_1, \ldots, t_n)]\!] = \mathcal{P}_{prog}\left[\!\!\left[\begin{array}{l} s = \texttt{vouterproduct}(t_1, \ldots, t_n) \\ t' = \texttt{eq}(s, \epsilon \rightarrow I') \end{array} \right]\!\!\right], \quad (35)$$

where $I' = I_1 \,||\, \ldots \,||\, I_n$, and the iterator lists $I_1, \ldots, I_n$ are pairwise disjoint, so that no iterator appears more than once in the concatenated list I'. Note that the implementation of **outerproduct** is not quite as modular as Equation (35) suggests: to construct the final iterator list I', the iterator lists $I_1, \ldots, I_n$ from the definition of **vouterproduct** are needed.

5 Evaluation of TeML

Table 1 describes a range of common tensor kernels from different application domains. These kernels serve as benchmarks for our evaluation of TeML that assess three different aspects of the language:

1. The capability to express tensor computations efficiently. Here we compare with TensorFlow [2], whose abstractions for tensor expressions are similar to TeML's tensor operations from Section 4.2.
2. The ability to reproduce loop optimization paths of existing tools. It is natural to compare with Pluto [8], which gives access to powerful polyhedral-based transformations through an interface allowing to enable or disable transformation heuristics.
3. The ability to easily extend optimization paths by composing with additional transformations, leading to the generation of C programs that outperform the ones generated with Pluto.

As this section unfolds, it will become clear that not all kernels from Table 1 can be meaningfully used in the evaluations of all of the three aspects.

All experiments reported in this section were performed on an Intel(R) Core(TM) i7-4910MQ CPU (2.90GHz, 8 hyperthreads, 8192KB of shared L3 cache) running the Ubuntu 16.04 operating system. The generated C programs (either from TeML or Pluto) were compiled with the Intel C compiler ICC 18.02 using the flags `-O3 -xHost -qopenmp`. We used Pluto version 0.11.4, and TensorFlow version 1.6 with support for AVX, FMA, SSE, and multi-threading.

5.1 Expressing Tensor Computations

Most benchmark kernels can be implemented using TeML's more abstract tensor operations, which keeps the kernel code short. As shown in Table 1, TensorFlow and TeML programs are of comparable sizes in terms of lines of code (LOC).

Most TensorFlow kernels use either **einsum** or **tensordot**. The latter is the equivalent of TeML's **contract**. TensorFlow's **einsum** operation is more low-level than **tensordot** and can thus be used when the semantics of **tensordot** are too restrictive, e.g. in batched matrix multiplication. In TeML, the same functionality can be implemented using also low-level operations, i.e. **add** and **mul**. Batched matrix multiplication ($C = A\,B$) must be built with more low-level operations in both frameworks due to the batch index b:

```
C[b][i][j] += A[b][i][k] * B[b][k][j]
```

Note that **einsum** is not flexible enough to implement all kernels that can be expressed in TeML, as evidenced by the convolution kernel *gconv* and the stencil kernel *blur* in Table 1. TensorFlow does also not have a dedicated construct for outer products, but its documentation explicitly recommends using **einsum** for this purpose.[1]

In summary, practically all of TensorFlow's constructs for tensor expressions have equivalents in TeML that can be used as effectively. TeML also offers support for expressing computations that cannot be implemented in TensorFlow.

5.2 Reproducing Pluto's Optimization Paths

For the tensor kernels from Table 1 we have identified the fastest program variants that can be generated with Pluto by manipulating its heuristics for loop fusion, tiling, interchange, vectorization, and thread parallelism.[2] Table 2 lists the TeML equivalents of the loop optimization paths that caused Pluto to generate the fastest programs. Note that the TeML transformations **vectorize** and **parallelize** have been implemented with compiler-specific pragmas for vectorization and OpenMP pragmas for thread parallelization.

The stencil kernel *blur* is not included in Table 2 since Pluto's best optimization path for this kernel performs loop skewing, which cannot yet be expressed in TeML. Also, as standard matrix multiplication has been thoroughly studied, cf. [34] in particular, our analysis focuses on *bmm*, which presents a less conventional computation pattern.

Since the sequences of TeML loop transformations in Table 2 reproduce the effects of Pluto's optimizations, C programs generated either from TeML or Pluto for the kernels listed in Table 2 have equal execution times. For space reasons we have omitted plots of these execution times since they would only show relative speed-ups of 1.0× (within measurement accuracy) between Pluto and TeML.

5.3 Performance Comparison

We now study opportunities for composing optimization paths with TeML that lead to generated C programs that outperform programs generated with Pluto. For completeness we also indicate the performance of the corresponding TensorFlow kernels. Figure 11 shows speed-ups relative to the sequential C implementation that is the starting point for applying optimizations with Pluto. Speed-ups are shown for the best program variants generated with Pluto and TeML, and for TensorFlow kernels whenever they exist (cf. Table 1).

Pluto's best variants for *mttkrp*, *bmm*, and *sddmm* still offer opportunities for data transposition, which can significantly

[1] https://www.tensorflow.org/api_docs/python/tf/einsum

[2] Note that Pluto offers only limited supported for unrolling heuristics.

Table 1. Tensor kernels implemented in TensorFlow and TeML. Application domains: Linear Algebra (LA), Deep Learning (DL), Machine Learning (ML), Data Analytics (DA), Fluid Dynamics (FD), Image Processing (IP).

	Name	Domain	TensorFlow		TeML	
			LOC	Constructs used	LOC	Constructs used
Matrix Multiplication	*mm*	LA	3	matmul	3	contract
transposed	*tmm*	DL	3	matmul:transpose=True	4	transpose, contract
batched	*bmm*	DL	3	einsum	3	mul, add
Grouped Convolutions [38]	*gconv*	DL	N/A	Not implemented. Incompatible with einsum.	5	vmul, add
Matricized Tensor Times Khatri-Rao product	*mttkrp*	DA	4	einsum or tensordot, multiply	5	vcontract, contract
Sampled Dense-Dense Matrix Product	*sddmm*	ML	4	einsum or tensordot, multiply	6	vcontract, entrywise_mul
Interpolation [17]	*interp*	FD	3	einsum or tensordot	5	contract
Helmholtz [17]	*helm*	FD	N/A	Required division not well supported.	9	contract, outerproduct, div, entrywise_mul
Blur	*blur*	IP	N/A	No stencil support.	9	op, vop
Coarsity [15]	*coars*	IP	6	einsum or multiply, subtract	6	ventrywise_mul, entrywise_sub

Table 2. Equivalents of Pluto's optimization paths in TeML. Kernel data sizes in parentheses.

*mttkrp (250*250*250)*	*sddmm (4096*4096)*	*bmm (8192*72*26)*	*gconv (32*32*32*32*7*7)*	*interp (50000*7*7*7)*	*helm (5000*13*13*13)*	*coars (4096*4096)*
parallelize(l, 1) interchange(l, 2, 3)	interchange(l, 2, 3), parallelize(l, 1), vectorize(l, 3)	tile(l, 32) interchange(l, 7,8) parallelize(l, 1) vectorize(l, 8)	interchange(l1, 4, 5) interchange(l1, 5, 6) parallelize(l1, 1) vectorize(l1, 9) parallelize(l2, 1) vectorize(l2, 9)	interchange(l1, 4, 5), vectorize(l1, 5), interchange(l2, 4, 5), vectorize(l2, 5), parallelize(l1, 1), parallelize(l2,1), parallelize(l3, 1)	fuse_outer(l4, l5, 5), fuse_outer(l4, l6, 5), parallelize(l1, 1), parallelize(l2, 1), parallelize(l3, 1), parallelize(l4, 1), vectorize(l1, 2), vectorize(l2, 3), vectorize(l3, 4)	tile(l, 32) parallelize(l, 1) vectorize(l, 4)

improve performance if copy overheads are negligible. The relevant loop nest in the *mttkrp* kernel is this:

```
for (int i = 0; i <= (I-1); i++)
  for (int j = 0; j <= (J-1); j++)
    for (int k = 0; k <= (K-1); k++)
      for (int l = 0; l <= (L-1); l++)
        A[i][j] = B[i][k][l] * D[l][j] * C[k][j];
```

The loop interchange $j \leftrightarrow l$ would eliminate the column-major access of D, and $j \leftrightarrow k$ would eliminate that of C. Both column-major accesses cannot be eliminated at the same time (without negatively affecting the access patterns of A and B). Pluto chooses to interchange $j \leftrightarrow k$ (cf. Table 2), which only eliminates the column-major access of C. The following TeML meta-program implements *mttkrp* and resolves the column-major access of D by means of data transposition, yielding a speed-up of 1.74× compared to Pluto, with the cost of transposition included (Figure 11a).

```
B = tensor(double, [250, 250, 250])
C = tensor(double, [250, 250])
D = tensor(double, [250, 250])
Dt = transpose(D, [[1, 2]])
tmp = vcontract(B, Dt, [3, 1])
A = contract(tmp, C, [2, 1])
ld = build(Dt)
l = build(A)
l1 = parallelize(l, 1)
l2 = interchange(l1, [2, 3])
```

For *bmm*, however, when executing on more than two cores, the copy overhead that results from the transposition loop is greater than the cost of transposed data accesses, making it difficult to outperform Pluto (Figure 11b).[3] Similarly, an additional transposition makes *sddmm*'s execution about 2× slower. When TeML does not apply the additional transposition to the *bmm* and *sddmm* kernels, performance is on par with the generated Pluto program as explained in Section 5.2. This shows that initially appealing transformations do not always guarantee a gain in performance.

It is known that BLAS implementations can outperform code generated with Pluto thanks to manually implemented optimizations that build on a detailed understanding of the underlying hardware. Some of the relevant transformations are readily available in TeML (e.g. loop tiling). As TeML is

[3]Note that previous work [34] produced similar results for standard matrix multiplication and Pluto.

designed to be easily extensible, we believe that it can be extended with missing transformations from which matrix multiplication will benefit further.

Sequences of contractions are central to both the *interp* and the *helm* kernel. In *interp*, the C implementation of the first contraction is as follows.

```
for (int i1 = 0; i1 <= (N-1); i1++)
  for (int i2 = 0; i2 <= (N-1); i2++)
    for (int i3 = 0; i3 <= (N-1); i3++)
      for (int i4 = 0; i4 <= (N-1); i4++)
        t[i1][i2][i3] += A[i1][i4] * u[i4][i2][i3];
```

Pluto chooses to permute the loops into `i1`, `i2`, `i4`, `i3` . However, all transposed accesses are eliminated if the loops are ordered into `i1`, `i4`, `i2`, `i3` with TeML. Figure 11e shows that this potentially yields slightly better performance. Applying the analogous permutation to *helm* inhibits the fusions that Pluto chooses to apply (cf. Table 2) but leads to noticeably better performance than Pluto's fusions (cf. Figure 11f).

Loop unrolling considerably speeds up *gconv* (Figure 11d) as small inner dimensions lend themselves well to full unrolling. While Figure 11d looks promising, one has to be cautious when comparing with Pluto since it cannot apply an unrolling heuristic analogous to TeML.

For the *coars* kernel we were unable to identify transformations that outperform Pluto's heuristics.

It should be noted that it is not fair to compare the performance results for Pluto and TeML directly with those for TensorFlow in Figure 11. Unlike Pluto and TeML, TensorFlow does not let programmers flexibly configure the loop optimizations that are applied to a kernel. Furthermore, since TensorFlow is targeted at the machine learning domain, it is not reasonable to expect it to perform well at optimizing kernels from other application domains, e.g. *interp*.

6 Related Work

Many frameworks for handling tensor computations exist, ranging from general-purpose [16, 18, 28–31] to domain-specific solutions [3, 6, 11, 20, 22, 26, 38]. TeML distinguishes itself from these frameworks in two major ways.

First, the domain-specific solutions offer abstractions for tensor algebra that are appropriately chosen for the respective application domain. TeML's representation of tensor expressions is purposefully more low-level in the sense that it does not include abstractions for mathematical structures that are relevant in any specific domain. This makes TeML more flexible and enables its use as an intermediate language that gives compiler implementers direct access to powerful capabilities for meta-programming and composing optimizations for different domains and platforms.

Second, the existing general-purpose and domain-specific solutions both rely on built-in heuristics for transforming input porgrams into optimized code. Users can only crudely manipulate these heuristics, if at all. For instance, the Tensor Contraction Engine [6] within the NWChem system [36] is a tool that implements fixed loop transformation heuristics for applications in quantum chemistry. Analogously, Coffee [20] is a compiler that interfaces with PyOP2 [24] and implements optimization heuristics tailored to finite element problems. TeML, on the other hand, is more general in that it lets users directly specify and manipulate code optimization strategies by composing transformations at the meta-level.

The TACO [18] framework hits an interesting point in the design space of tensor languages. TACO does not target a specific application domain, but as a language and compiler for kernels that mix dense and sparse tensor algebra it focuses on the integration of different data structures for representing tensors internally. To handle the mix of dense and sparse structures efficiently, TACO also employs suitable built-in strategies. Again this means that, unlike in TeML, code transformation and optimization cannot be controlled by user input. Furthermore, TACO does not employ classic loop transformations as found, for instance, in polyhedral frameworks such as Pluto. Hence, TACO's transformations do not directly compare to the ones provided by TeML.

TeML's level of abstraction is similar to those of TensorFlow [2], Theano [7], Numpy [1], or SaC [27], while also capturing explicit transformations available in meta-programming languages [4, 9, 10, 12, 13, 19, 21, 23, 35, 39]. In terms of design, TeML is much closer to Halide [23] and TVM [10], with differences in expressiveness and programming style. Identifying tensor operation patterns requires additional program analysis in Halide and TVM whereas the patterns of higher-level operations are native to TeML. Also, Halide and TVM decouple the specification of computation (which is functional) from the specification of transformations (which is imperative). TeML, however, unifies both stages following the same functional style, which has two advantages. First, considering tensor operation patterns as native lets TeML directly understand algebraic properties, which is necessary to enable algebraic transformations. Second, generating different program variants does not require different meta-programs. In TeML, a single meta-program can include the specification of one tensor computation followed by different branches of transformation paths.

Very few existing frameworks and tools come with formal semantics. One exception is Lift [33], a functional language from which optimized and portable GPU code can be generated. The definition of Lift's core language is given in terms of denotational semantics [32]. However, Lift's approach to abstracting computations and transformations differs from TeML's: computations are expressed using combinators, and rewrite rules are used to transform programs. The meta-programming tools Clay [4], URUK [12], CHiLL [9], and Loo.py [19] rely largely on the polyhedral formalism, which focuses on the representation, analysis, and transformation of loops. However, none of these tools have semantics that model loop transformations. To the best of our knowledge,

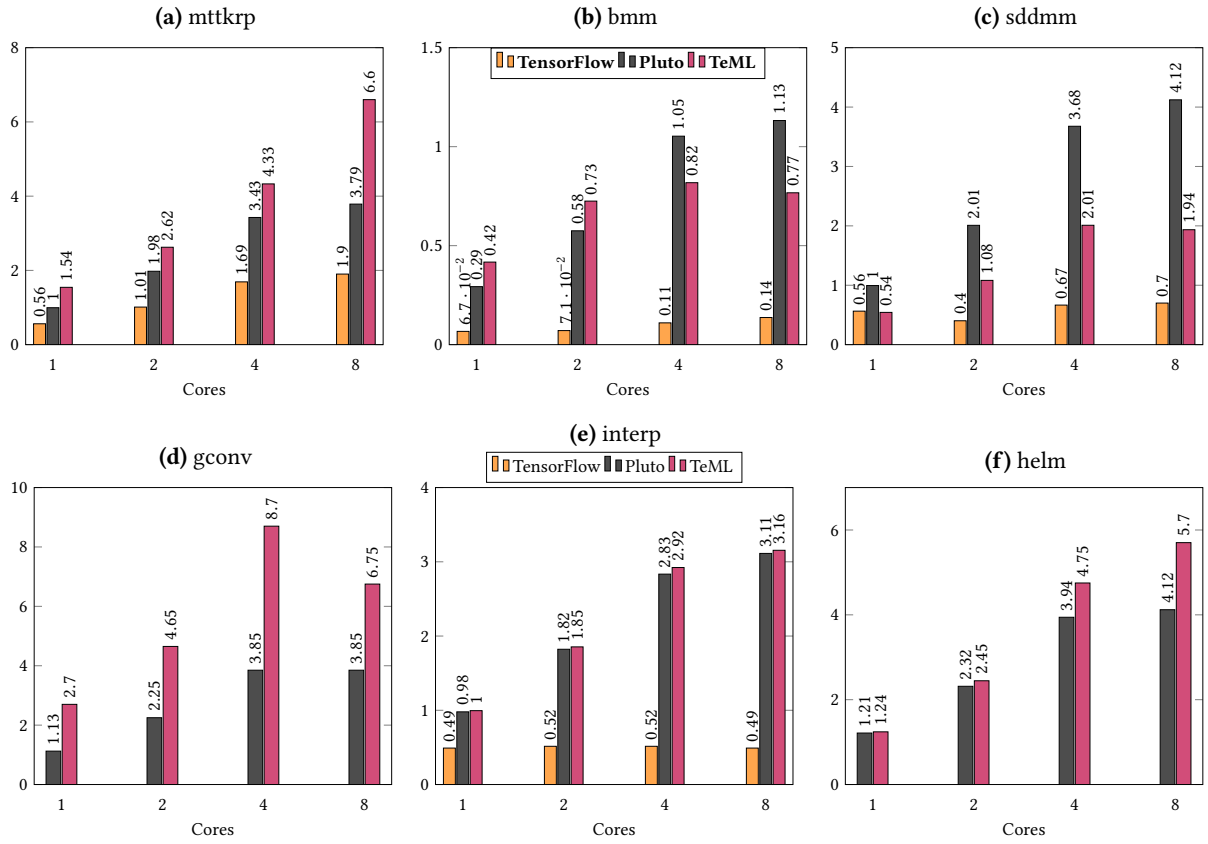


Figure 11. Speed-ups relative to sequential (i.e. single-core) C implementations.

TeML is the first (meta-)language that comes with denotational semantics for both tensor computations and explicit transformations thereof.

7 Summary and Outlook

We have presented TeML, a meta-language for programming and optimizing tensor computations. TeML offers similar levels of abstraction for expressing tensor computations as existing tensor languages, e.g. TensorFlow and Theano. Additionally, loop transformations are first class citizens in TeML, enabling meta-programming of optimization schemes comparable to and better than those of Pluto, a state-of-art polyhedral compiler, in only a few lines of code. The semantics of TeML have been formally specified in a denotational style, exhibiting a high level of compositionality as more abstract tensor operations and loop transformations can be formally defined by composition of more fundamental ones.

In the future, it would be interesting to augment TeML with abstractions for memory virtualization and mapping in order to handle, e.g., NUMA and distributed memory placement, data alignment, and padding. The semantics would have to be extended accordingly.

Note that TeML currently has no notion of whether loop transformations preserve the semantics of target language programs. This is on purpose: past research on affine transformations [5, 14, 37] has highlighted advantages of allowing non-semantics-preserving transformations as intermediate steps in optimizing programs so long as the full optimization process, once completed, retains the original meanings of programs. Extending TeML with facilities for both requiring and guaranteeing that certain transformations preserve program semantics is an interesting direction for future work.

While the present work has focused on the design of TeML as a meta-language, future work could assess the possibility of using TeML to identify platform-independent or portable optimization strategies, targeting also GPU platforms.

Finally, to enhance the usability of TeML in certain application domains, future work should aim at identifying suitable high-level abstractions for stencil patterns and general convolutions. In a similar vein, options for supporting sparse tensors might be worth investigating.

Acknowledgments

This work was partially supported by the German Research Council (DFG) through the Cluster of Excellence 'Center for

Advancing Electronics Dresden' (cfaed) and by PSL University through the ACOPAL project.

References

[1] 2017. NumPy, package for scientific computing with Python. http://www.numpy.org/.

[2] Martín Abadi and Ashish Agarwal et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. http://download.tensorflow.org/paper/whitepaper2015.pdf.

[3] Martin S. Alnaes, Anders Logg, Kristian B. Olgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations. *ACM Trans. Math. Softw.* 40, 2, Article 9 (March 2014), 37 pages. https://doi.org/10.1145/2566630

[4] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[5] Cédric Bastoul and Paul Feautrier. 2004. More Legal Transformations for Locality. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31-September 3, 2004, Proceedings*. 272–283. https://doi.org/10.1007/978-3-540-27866-5_36

[6] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, Xiaoyang Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, Chi chung Lam, Qingda Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proc. IEEE* 93, 2 (Feb 2005), 276–292. https://doi.org/10.1109/JPROC.2004.840311

[7] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[9] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Technical Report 08-897, University of Southern California.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[11] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/2254064.2254079

[12] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the Search for Compositions of Program Transformations. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/1088149.1088169

[13] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. 2006. *A Language for the Compact Representation of Multiple Program Versions.* Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151.

[14] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[15] Olfa Haggui, Claude Tadonki, Lionel Lacassagne, Fatma Sayadi, and Bouraoui Ouni. 2018. Harris corner detection on a NUMA manycore. *Future Generation Computer Systems* (2018). https://doi.org/10.1016/j.future.2018.01.048

[16] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[17] Immo Huismann, Jörg Stiller, and Jochen Fröhlich. 2016. *Fast Static Condensation for the Helmholtz Equation in a Spectral-Element Discretization.* Springer International Publishing, Cham, 371–380.

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[19] Andreas Klöckner. 2014. Loo.Py: Transformation-based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 82, 6 pages. https://doi.org/10.1145/2627373.2627387

[20] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2015. Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly. *ACM Trans. Archit. Code Optim.* 11, 4, Article 57 (Jan. 2015), 25 pages. https://doi.org/10.1145/2687415

[21] Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, and Bernd Trenkler. 2004. Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests. In *Euro-Par 2004 Parallel Processing*, Marco Danelutto, Marco Vanneschi, and Domenico Laforenza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–81.

[22] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (Feb 2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[24] Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicolas Loriant, David A. Ham, Carlo Bertolli, and Paul H. J. Kelly. 2012. PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC '12)*. IEEE Computer Society, Washington, DC, USA, 1116–1123. https://doi.org/10.1109/SC.Companion.2012.134

[25] Norman A. Rink. 2018. Modeling of languages for tensor manipulation. *CoRR* abs/1801.08771 (2018). arXiv:1801.08771 http://arxiv.org/abs/1801.08771

[26] Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. 2018. CFDlang: High-level Code Generation for High-order Methods in Fluid Dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. ACM, New York, NY, USA, Article 5, 10 pages.

https://doi.org/10.1145/3183895.3183900

[27] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[28] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-scale Linear Algebra Applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 327–339. https://doi.org/10.1145/3168812

[29] Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *International Symposium on Code Generation and Optimization (CGO)*. 117–127.

[30] Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR* abs/1607.00145 (2016). http://arxiv.org/abs/1607.00145

[31] Paul Springer, Aravind Sankaran, and Paolo Bientinesi. 2016. TTC: A Tensor Transposition Compiler for Multiple Architectures. *CoRR* abs/1607.01249 (2016). http://arxiv.org/abs/1607.01249

[32] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[33] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 74–85. http://dl.acm.org/citation.cfm?id=3049832.3049841

[34] Adilla Susungi, Albert Cohen, and Claude Tadonki. 2017. More Data Locality for Static Control Programs on NUMA Architectures. In *Proceedings of the 7th International Workshop on Polyhedral Compilation Techniques (IMPACT '17)*.

[35] Adilla Susungi, Norman A. Rink, Jerónimo Castrillón, Immo Huismann, Albert Cohen, Claude Tadonki, Jörg Stiller, and Jochen Fröhlich. 2017. Towards Compositional and Generative Tensor Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 169–175. https://doi.org/10.1145/3136040.3136050

[36] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477 – 1489. https://doi.org/10.1016/j.cpc.2010.04.018

[37] Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. 2007. Automatic Correction of Loop Transformations. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*. 292–304. https://doi.org/10.1109/PACT.2007.17

[38] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[39] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370637