Towards Compositional and Generative Tensor Optimizations

Adilla Susungi*
MINES ParisTech
PSL Research University
France

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

Immo Huismann
Chair of Fluid Mechanics
Technische Universität Dresden
Germany

Albert Cohen
INRIA
École Normale Supérieure
France

Claude Tadonki
MINES ParisTech
PSL Research University
France

Jörg Stiller
Jochen Fröhlich
Chair of Fluid Mechanics
Technische Universität Dresden
Germany

Abstract

Many numerical algorithms are naturally expressed as operations on tensors (i.e. multi-dimensional arrays). Hence, tensor expressions occur in a wide range of application domains, e.g. quantum chemistry and physics; big data analysis and machine learning; and computational fluid dynamics. Each domain, typically, has developed its own strategies for efficiently generating optimized code, supported by tools such as domain-specific languages, compilers, and libraries. However, strategies and tools are rarely portable between domains, and generic solutions typically act as "black boxes" that offer little control over code generation and optimization. As a consequence, there are application domains without adequate support for easily generating optimized code, e.g. computational fluid dynamics. In this paper we propose a generic and easily extensible intermediate language for expressing tensor computations and code transformations in a modular and generative fashion. Beyond being an intermediate language, our solution also offers meta-programming capabilities for experts in code optimization. While applications from the domain of computational fluid dynamics serve to illustrate our proposed solution, we believe that our general approach can help unify research in tensor optimizations and make solutions more portable between domains.

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

Keywords numerical methods, computational fluid dynamics (CFD), tensor methods, intermediate language, meta-programming, code generation and optimization

ACM Reference Format:
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 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3136040.3136050

1 Introduction

Many numerical applications spend a large fraction of their runtime evaluating complex tensor expressions. Developers, therefore, spend a lot of time optimizing code that evaluates tensor expressions. A number of domain-specific languages and libraries exist to assist developers and domain-experts [4, 6, 7]. However, such tools do not usually generalize well from one application domain to another. More generic solutions, on the other hand, may not provide adequate program transformations [3, 15, 16, 23]. This forces experts from many different domains to painstakingly hand-optimize the key loop nests in their numerical applications.

In this work, we consider examples from the domain of computational fluid dynamics (CFD), which is the study of fluid flows using numerical methods. Fluid flows are governed by the Navier-Stokes equations (NSE), solutions of which can be obtained by solving a succession of so-called

*Corresponding author: adilla.susungi@mines-paristech.fr
†Corresponding author: norman.rink@tu-dresden.de

GPCE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136050

Helmholtz equations. Hence, in implementing fast and efficient NSE solvers, it is crucial to focus on the Helmholtz operator. Using a tensor-based representation of the Helmholtz operator enables algebraic transformations that lead to considerably lower execution runtime when coupled with a well established CFD-related technique [13, 14, 19].

In a typical CFD application, the volume of interest is divided into thousands of mesh points. At each point of the mesh, successive computations are performed over small and dense tensors. Computations combine tensor contractions, entry-wise multiplications, and outer products. Since the solution process is iterative, computations are repeated for hundreds of thousands of time steps, leading to execution times of several weeks, even when parallelized across thousands of cores. This makes approaches such as iterative compilation [17] useful to find the best program variant.

Tensor contraction is also the central operation in quantum chemistry simulations. Machine learning applications, however, rely more on entry-wise operations, and contractions are usually limited to matrix-matrix or matrix-vector multiplication. While frameworks and libraries exist for handling tensor expressions in both domains [4, 6, 7], to the best of our knowledge, no such tools are available in the CFD domain. Thus, we see a general need for convenient ways to describe, manipulate, and optimize tensor expressions for the CFD domain and beyond.

We envision a programming flow as in Figure 1, where different languages are lowered to a tensor intermediate language. At this level, expressions can be easily manipulated and transformed by optimization experts using meta-programming techniques. Alternatively, a compiler can perform an iterative search for beneficial transformations.

Contribution. Building upon previous work [25], we propose a generative and extensible intermediate language with meta-programming capabilities. The language provides expressivity for describing (a) different types of tensor computations (Section 2) and (b) modular transformations that can be easily composed (Section 3). Inspired by approaches such as [9–11, 20], we separate the manipulation of data layouts, memory placements, access functions for tensors, and iteration domains (i.e. loop bounds). We can therefore modify, for instance, data layouts without impacting the entire program, apply transformations without increasing code complexity, and easily generate different program variants (Sections 4 and 5).

2 Tensor Operations and Expressions

This section introduces the relevant tensor operations and their representation in our intermediate language.

2.1 Tensor Operations

The data structure that underlies a tensor is an N-dimensional array. The array becomes a tensor when it is accompanied by certain algebraic operations. A 1-dimensional tensor is a vector, and a 2-dimensional tensor is a matrix; matrix-vector and matrix-matrix multiplication are special cases of the more general tensor contraction operation discussed below.

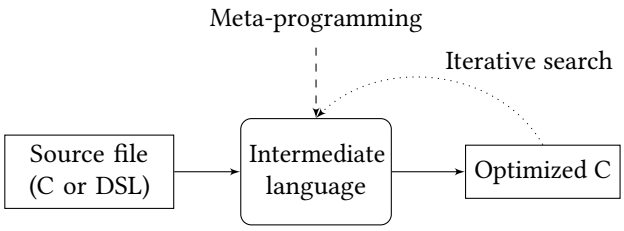


Figure 1. Envisioned tool flow.

In this paper, we use code examples and formulae from CFD applications, which typically describe phenomena in 3-dimensional space. Therefore, the following operations are introduced using 3-dimensional tensors A and B.

Contraction. If, say, the 3rd dimension of A and the 1st dimension of B have the same size d, a contraction along these dimensions is

$$C_{ijlm} = \sum_{k=1}^{d} A_{ijk} \cdot B_{klm} \,. \tag{1}$$

The resulting tensor C is 4-dimensional. From (1) it is clear that contraction generalizes matrix-matrix multiplication. Note, however, that one can contract along any pair of dimensions of A and B, not only the last and first respectively, as long as the contracted dimensions have the same size.

Outer product. The outer product of A and B produces a 6-dimensional tensor C such that

$$C_{ijklmn} = A_{ijk} \cdot B_{lmn} \,. \tag{2}$$

Entry-wise multiplication. The entry-wise tensor multiplication of A and B produces a 3-dimensional tensor C:

$$C_{ijk} = A_{ijk} \cdot B_{ijk} \,. \tag{3}$$

For matrices, entry-wise multiplication is also known as the Hadamard product. Although not needed in this work, all arithmetic operations have entry-wise extensions to tensors.

2.2 Expressing Tensor Computations

Our intermediate language purposefully relies on high-level constructs to facilitate its use for meta-programming.

A tensor T is declared as a multi-dimensional array:

▹ `T = array(N, dtype, [size`$_1$`, ..., size`$_N$`])` ,

where N is the number of dimensions, `dtype` is the data type of the array elements, and `[size`$_1$`, ..., size`$_N$`]` are the sizes of the dimensions.

Complex tensor expressions are typically composed of the previously introduced operations, i.e. contraction, outer

```
A = array(2, double, [N, N])
u = array(3, double, [N, N, N])

tmp1 = contract(A, u, [[2, 1]])
tmp2 = contract(A, tmp1, [[2, 2]])
v = contract(A, tmp2, [[2, 3]])
```

Figure 2. Intermediate language code for Equation (5).

```
A = array(2, double, [N, N])
u = array(3, double, [N, N, N])

tmp1 = outerproduct(A, A)
tmp2 = contract(A, u, [2, 1])
v = contract(tmp1, tmp2, [[2, 3],
                          [4, 2]])
```

Figure 3. Intermediate language code for Equation (6).

```
A = array(2, double, [N, N])
u = array(3, double, [N, N, N])

tmp1 = outerproduct(A, A)
tmp2 = contract(tmp1, u, [[2, 2],
                          [4, 1]])
v = contract(A, tmp2, [[2, 3]])
```

Figure 4. Intermediate language code for Equation (7).

product, and entry-wise arithmetic. Therefore, these operations are mapped directly to the following constructs in our intermediate language:

- ▹ `outerproduct(tensor1, tensor2)`,
- ▹ `contract(tensor1, tensor2, [[rank1i, rank2i], ..., [rank1n, rank2n]])`,
- ▹ `entrywise(tensor1, tensor2)`,

where, in the context of this paper, `entrywise` refers to multiplication. Supporting other arithmetic operations in the intermediate language is straightforward.

The semantics of `contract` and its arguments are as follows. The last argument is a list of pairs, in which each pair $[\texttt{rank}_{1_i}, \texttt{rank}_{2_i}]$ specifies that the dimension $\texttt{rank}_{1_i}$ of $\texttt{tensor}_1$ and $\texttt{rank}_{2_i}$ of $\texttt{tensor}_2$ are contracted together. If $\texttt{tensor}_1$ is N-dimensional, then $\texttt{rank}_{1_i}$ can take numerical values ranging from 1 to N, and analogously for $\texttt{tensor}_2$. Note in particular that the order of the first two arguments of `contract` matters since the ranks in the last argument are tied to either $\texttt{tensor}_1$ or $\texttt{tensor}_2$.

Simple, yet non-trivial, code examples in our intermediate language can be derived from *Interpolation*, a tensor expression that appears in CFD applications:

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn} \,. \tag{4}$$

Interpolation is used, for example, to facilitate a change of basis or low-pass filtering in numerical applications that utilize *spectral element methods* [12]. Adding parentheses to (4) enforces different evaluation orders:

$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \,, \tag{5}$$

$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \,, \tag{6}$$

$$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})) \,. \tag{7}$$

Figures 2–4 show the intermediate language code for Equations (5)–(7) respectively. Equations (6) and (7) each contain a term $(A \cdot A)$ that does not involve a contraction and therefore gives rise to the `outerproduct` in Figures 3 and 4 respectively. The different evaluation schemes in Figures 2–4 result in different performance characteristics of generated code. This is a general problem that we address by adding composable transformations to the intermediate language in Section 3.

2.3 Memory Model and Transpositions

The physical layout of a tensor in memory may differ from its logical structure determined by the ordering of its dimensions. In particular, new logical tensors can be created by transposing the dimensions of a tensor, which merely changes the order in which elements of the tensor are accessed and, hence, does not require that additional memory be allocated in the generated code. Transpositions appear frequently in tensor expressions, and whether it is beneficial to allocate memory for a transposed tensor depends on the specific memory access patterns for that tensor. Our intermediate language supports transposition into both *physical* and *virtual* tensors, where new memory is allocated only for physical tensors:

- ▹ `transpose(T, rank1, rank2)` for transposing the physical *or* virtual tensor `T` into a *physical* tensor,
- ▹ `vtranspose(T, rank1, rank2)` for transposing the physical *or* virtual tensor `T` into a *virtual* tensor.

Both operations produce a new tensor object in the intermediate language by swapping the dimensions $\texttt{rank}_1$ and $\texttt{rank}_2$ of tensor T. Note that neither operation has any side-effects on the original object T.

The transposition of a matrix A is expressed in the intermediate language by

```
B = transpose(A, 1, 2)
```

which creates a new physical matrix B that is backed by memory. For a more complex example, assume that A is a 4-dimensional tensor. The intermediate language statements

```
A1 = transpose(A, 1, 3)
B  = transpose(A1, 2, 4)
```

express the permutation $\{1 \leftrightarrow 3, 2 \leftrightarrow 4\}$ of the dimensions of A. The result of this permutation is stored in the physical tensor B.

Since virtual tensors are not backed by memory, they are views of physical tensors that can be inserted at zero cost. Their main purpose is to insert a layer of abstraction before

physical indexing, effectively decoupling access functions from the physical memory layout of tensors. This, in turn, enables independent reasoning and transformation of access functions, data layout, and iteration domains.

2.4 Iterators

The tensor operations contraction, outer product, and entry-wise multiplication implicitly define loops over the dimensions of the tensors involved in these operations. To enable transformations of iteration domains, one requires handles on the iteration domains of the implicitly defined loops. To this end, one first defines an iterator

```
i = iterator(0, N, 1)
```

which, in this case, ranges from 0 to $N-1$ (inclusive), at a step size of 1. Then, the iterators involved in the iteration domain over an N-dimensional tensor `T` are made explicit:

▷ `build(T, [iterator`$_1$`, ..., iterator`$_N$`])`

Figure 5 shows a specific example of how the relevant iterators and iteration domain are defined for the first contraction from Figure 2.

```
tmp1 = contract(A, u, [[2, 1]])

i = iterator(0, N, 1)
m = iterator(0, N, 1)
n = iterator(0, N, 1)
l = iterator(0, N, 1)

build(tmp1, [i, m, n, l])
```

Figure 5. Iteration domain for a contraction.

The first three iterators `i`, `m`, `n` span the dimensions of the tensor `tmp1` that results from the contraction. The last iterator `l` corresponds to the contracted dimensions.

In our intermediate language, transformations can act on iteration domains that are defined by sets of iterators. See examples in Section 3.

3 Compositions of Transformations

Having described the relevant tensor operations and how they are represented in our intermediate language, we now turn to the composition of transformations.

3.1 Transformations

The loops that are implicitly defined by the tensor operations from Section 2.2 can be transformed by operating on declared iterators. Recall from Section 2.4 that iterators are tied to tensors through the `build` directive. In our intermediate language, loop transformations are applied by a statement of the following form:

▷ `transformation_name(iterator`$_1$ `[, iterator`$_2$`, ..., iterator`$_N$`][, param`$_1$`, ..., param`$_N$`])`

A transformation can either involve a single or multiple iterators, and additional parameters (e.g. unroll factor), which are optional. In contrast to tensor operations, transformations do not return new iterators, which would unnecessarily increase the difficulty of loop management when considering transformations such as loop distribution and fusion. Instead, transformations have side-effects on their iterator arguments. Side-effects can consist of modifying

1. an iterator's loop bounds,
2. an iterator's parallel specification,
3. statements in which the iterator is used, and
4. the validity of the iterator, if the iterator is deleted by the applied transformation, e.g. by loop fusion.

Loop interchange, fusion, and parallelization are transformations that are frequently used in improving the performance of CFD applications. As an example, consider two loop nests each of which consists of two loops. Let the iterators `i1` and `i2` define the iteration domain of the first loop nest, and let iterators `i3` and `i4` define the second loop nest. By stating

```
interchange(i1, i2)
fuse(i2, i3)
fuse(i1, i4)
parallelize(i2, THRD, None, [i1])
```

in the intermediate language, code generation is directed to

1. swap the loops in the first loop nest (i.e. the loops associated with `i1` and `i2`),
2. completely fuse both loop nests, and
3. parallelize the resulting outer loop (traversed by `i2`) using threads.

The parameter `None` of `parallelize` implies that no thread scheduling is defined. (Other policies could be passed, for instance, to the OpenMP clause `schedule`.) The last parameter, `[i1]`, indicates that `i1` is the only private variable. Since the loop fusions have merged `i3` into `i2` and `i4` into `i1`, no further transformations involving `i3` and `i4` can be applied. Sometimes it is beneficial to replace a tensor with, say, its virtual transpose. This is achieved by the transformation

▷ `replace_array(iterator, old, new)`

which replaces the tensor `old` with the tensor `new` in the statement enclosed by a loop over `iterator`.

3.2 Implementation and Code Generation

In lowering intermediate language programs to C code, transformations are applied in their order of appearance. To this end, an intermediate language program is first translated into a data structure holding lists of tensors, iterators, loops, and transformations, each with respective attributes. More precisely:

1. tensors are classified as either physical or virtual;
2. iterators are built according to their specification;
3. transformations are identified and added to a list in their strict order of appearance in the program.

When the data structure is fully formed, transformations in the list are successively applied. This may modify iterator specifications and access functions and may create new loops. Code is then generated by simply translating the resulting, modified data structure into its C equivalent.

4 Step-by-step Example

In this section we implement the *Inverse Helmholtz* operator in our intermediate language, and we define transformations to optimize loop orders and array accesses. *Inverse Helmholtz* is a CFD kernel that consists of contractions with transposed accesses, an entry-wise multiplication, and contractions without transposed accesses:

$$t_{ijk} = \sum_{l,m,n} A^T_{kn} \cdot A^T_{jm} \cdot A^T_{il} \cdot u_{lmn} \tag{8}$$

$$p_{ijk} = D_{ijk} \cdot t_{ijk} \tag{9}$$

$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot p_{lmn} \tag{10}$$

Step 1: Declaration of tensors. We declare A, u, and D.

```
A = array(2, double, [N, N])
u = array(3, double, [N, N, N])
D = array(3, double, [N, N, N])
```

We prepare the transposed view of A using a virtual transposition to express A^T.

```
At = vtranspose(A, 1, 2)
```

Having declared the arrays, we proceed with the tensor computation. The contractions are decomposed as in Equation (5). We also express the entry-wise multiplication between D and $tmp3$ (see Equation (9)), where $tmp3$ is the last intermediate value of the first set of contractions.

```
tmp1 = contract(At, u, [2, 1])
tmp2 = contract(At, tmp1, [2, 2])
tmp3 = contract(At, tmp2, [2, 3])
tmp4 = entrywise(D, tmp3)
tmp5 = contract(A, tmp4, [2, 1])
tmp6 = contract(A, tmp5, [2, 2])
v = contract(A, tmp6, [2, 3])
```

Later we might want to apply transpositions to the first set of contractions, which would lead to transposed accesses of D. Therefore, we already declare a tensor Dt that is a transposed version of D.

```
Dt = transpose(D, 1, 3)
```

The overhead resulting from copying D into Dt can potentially be reduced by loop fusion.

Step 2: Declaring iterators. To be able to build all loops, we declare 30 unique iterators:

1. $td1$, $td2$, $td3$ for the transposition of D into Dt;
2. $i1$, $i2$, $i3$, $i4$ for the contraction into $tmp1$;
3. $j1$, $j2$, $j3$, $j4$ for the contraction into $tmp2$;
4. $k1$, $k2$, $k3$, $k4$ for the contraction into $tmp3$;
5. $l1$, $l2$, $l3$ for the entry-wise multiplication into $tmp4$;
6. $i12$, $i22$, $i32$, $i42$ for the contraction into $tmp5$;
7. $j12$, $j22$, $j32$, $j42$ for the contraction into $tmp6$;
8. $k12$, $k22$, $k32$, $k42$ for the contraction into v;

```
i1 = iterator(0, N, 1)
i2 = iterator(0, N, 1)
# ... other iterator declarations
```

Step 3: Expressing the loops. We then build the loops corresponding to the computations.

```
build(Dt, [td1, td2, td3])
build(tmp1, [i1, i2, i3, i4])
## Also applies to tmp2, ..., tmp6
build(v, [k12, k22, k32, k42])
```

Step 4: Applying transformations. We now apply transformations. Several dimensions can be interchanged in the loops computing $tmp1$, $tmp2$, $tmp3$, $tmp4$, and $tmp5$. In addition, `vtranspose` transforms the access functions of $tmp2$, $tmp3$, $tmp4$, and $tmp5$. Using `replace_array`, the resulting virtual tensors (i.e. $tmp2t$, $tmp3t$, $tmp4t$, and $tmp5t$) replace their un-transposed counterparts where desired.

```
interchange(i4, i3)
interchange(i4, i2)
interchange(j2, j1)
interchange(j1, j4)
tmp2t = vtranspose(tmp2, 1, 2)
replace_array(j3, tmp2, tmp2t)
replace_array(k4, tmp2, tmp2t)
tmp3t = vtranspose(tmp3, 1, 3)
replace_array(k4, tmp3, tmp3t)
interchange(k3, k2)
interchange(k1, k3)
interchange(k1, k2)
interchange(k1, k4)
replace_array(l3, D, Dt)
replace_array(l3, tmp3, tmp3t)
tmp4t = vtranspose(tmp4, 1, 3)
replace_array(l3, tmp4, tmp4t)
interchange(l3, l1)
replace_array(i42, tmp4, tmp4t)
tmp5t = vtranspose(tmp5, 2, 3)
interchange(i32, i22)
replace_array(j42, tmp5, tmp5t)
# ... parallelizations
```

5 Assessing Optimization Variants

We now assess different optimization variants of the *Interpolation* and *Inverse Helmholtz* operators. Since *Interpolation* appears as a core kernel inside *Inverse Helmholtz*, we study variants of *Interpolation* first. Programs are executed on a 24-core Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz (Haswell) using the Intel C compiler (version 13.0.1). Compilation options are `-O3 -march=native -openmp`, and vectorization is also enabled.

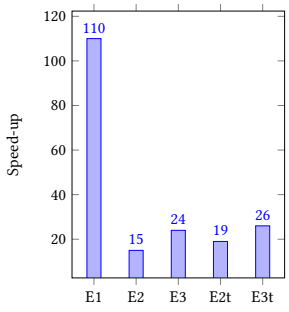
Interpolation. Starting with a naive evaluation of the tensor contractions, i.e. without any smart reordering of operations to reduce the algorithmic complexity, parallelization of the mesh loop across 24 cores leads to a speed-up of 20×. In the following, this parallelized naive evaluation serves as our baseline. Figure 6a shows the relative speed-ups compared to this baseline for parallelized implementations of the three *Interpolation* variants from Section 2.2. E1, E2, E3 respectively correspond to Equations (5), (6), and (7).

We attempted to reduce the execution times of E2 and E3 using additional loop fusions and permutations, producing the variants E2t and E3t. However, E1 remains the best CPU-based variant as it best minimizes the algorithmic complexity. A study on other architectures is a matter of future work.

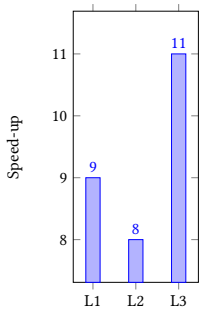
Inverse Helmholtz. Based on the observations in the previous paragraph, we evaluate the contractions in *Inverse Helmholtz* according to Equation (5). The sequential version of this serves as our baseline, and we assess the speed-ups for three strategies for dealing with column-major accesses:

1. L1 relies exclusively on loop interchanges. This is not sufficient to discard all existing column-major accesses, but it does not introduce additional copies.
2. L2 relies mainly on loop interchanges and introduces an additional copy for transposing A^T. This is also not sufficient to discard all column-major accesses.
3. L3 combines the previous strategy with implicit transpositions (i.e. not requiring additional loops) to completely discard all column-major accesses. This corresponds to the implementation presented in Section 4.

Figure 6b shows the relative speed-ups. Loop interchanges and implicit transpositions in L3 yield big enough performance gains to ameliorate the cost of explicitly copying D.



(a) *Interpolation*: speed-ups of code variants compared to naive evaluation (13.27s).



(b) *Inverse Helmholtz*: speed-ups compared to sequential execution (3.32s).

Figure 6. Speed-ups for *Interpolation* and *Inverse Helmholtz*. Mesh size = 750, data size = 33.

6 Related Work

In the present work, CFD applications served as examples to illustrate our intermediate language. The Firedrake [21] framework for CFD simulations allows high-level specification of partial differential equations and hides the details of efficient compilation from the user. By contrast, our proposed intermediate language is domain-agnostic and can thus be used to build abstractions for tensor optimizations into frameworks for different application domains.

Previous work [5, 8, 10, 11, 18, 22, 24] shows the need for approaches that ease the exploration of optimization search spaces. More specifically, Cohen et al. [10, 11] advocate the clear separation between the transformations of loop bounds, schedules, and access functions. Similarly, Halide [20] separates the manipulation of algorithms from that of schedules. Inspired by these approaches, we have provided the building blocks for flexibly composing computations and transformations, including memory placement modifications if needed [25]. This flexibility differentiates the presented work from existing solutions for optimizing tensor algebra. Whether generic or domain-specific, existing solutions are mainly black-boxes with little control over how transformations are applied [3, 4, 6, 7, 15, 16, 23].

The frameworks Theano, TensorFlow, and Numpy [1, 4, 7] provide a `tensordot` operator and thus also some flexibility in composing tensor computations. While the popular TensorFlow framework eases the implementation of full machine learning applications, it does not offer the kind of optimizations we seek. The XLA compiler intends to address this, but it is still in an experimental state [2]. The flexibility provided by our intermediate language also distinguishes our approach from the Tensor Contraction Engine (TCE) [6]. In TCE, which is designed for optimizing quantum chemistry applications, the search for the optimal decomposition of tensor computations is performed at the algebraic level before loop transformations are applied. Our approach offers the opportunity to assess the effects of loop transformations resulting from different decomposition strategies.

7 Conclusion

We have proposed an intermediate language for optimizing tensor computations. The language treats tensor computations and optimizations as modular building blocks. This enables easy assessment of code variants generated by composing different optimizations. We have demonstrated support for a class of applications in computational fluid dynamics for which existing solutions do not lead to satisfactory optimization results. However, we believe that the intermediate abstraction level makes our contribution suitable for tensor optimization across a wide range of domains. We intend to study the wider applicability of our approach, e.g. in the domain of machine learning. Another subject of future work is the automated search for good compositions of transformations and evaluation orders, and improving such a search by relying on domain-specific knowledge.

Acknowledgments

This work was partially funded by the German Research Council (DFG) through the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed) and by PSL Research University through the ACOPAL project.

References

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

[2] 2017. XLA: Accelerated Linear Algebra. https://www.tensorflow.org/performance/xla/. (2017).

[3] 2017. Xtensor, Multi-dimensional arrays with broadcasting and lazy computing. https://github.com/QuantStack/xtensor. (2017).

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

[5] 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

[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] 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.

[9] Albert Cohen, SÃľbastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. 2006. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming* 62, 1 (2006), 25–46. https://doi.org/10.1016/j.scico.2005.10.013 Special Issue on the First MetaOCaml Workshop 2004.

[10] Albert Cohen, Sylvain Girbal, and Olivier Temam. 2004. *A Polyhedral Approach to Ease the Composition of Program Transformations*. Springer Berlin Heidelberg, Berlin, Heidelberg, 292–303.

[11] 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

[12] M. O. Deville, P. F. Fischer, and E. H. Mund. 2002. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, Cambridge. https://doi.org/10.1017/cbo9780511546792

[13] 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.

[14] Immo Huismann, Jörg Stiller, and Jochen Fröhlich. 2017. Factorizing the factorization — a spectral-element solver for elliptic equations with linear operation count. *J. Comput. Phys.* 346 (2017), 437–448. https://doi.org/10.1016/j.jcp.2017.06.012

[15] Khaled Z. Ibrahim, Samuel W. Williams, Evgeny Epifanovsky, and Anna I. Krylov. 2014. Analysis and tuning of libtensor framework on multicore architectures. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*. 1–10.

[16] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. *The Tensor Algebra Compiler*. Technical Report. Massachusetts Institute of Technology.

[17] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. 2002. Embedded Processor Design Challenges. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Iterative Compilation, 171–187. http://dl.acm.org/citation.cfm?id=765198.765209

[18] 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

[19] Zu-Qing Qu. 2004. *Static Condensation*. Springer London, London, 47–70.

[20] 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

[21] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.* 43, 3, Article 24 (Dec. 2016), 27 pages. https://doi.org/10.1145/2998441

[22] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. *A Programming Language Interface to Describe Transformations and Code Generation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 136–150.

[23] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: A High-performance Tensor Transposition C++ Library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2017)*. ACM, New York, NY, USA, 56–62. https://doi.org/10.1145/3091966.3091968

[24] 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

[25] 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)*.