The Tensor Algebra Compiler

FREDRIK KJOLSTAD, Massachusetts Institute of Technology, USA
SHOAIB KAMIL, Adobe Research, USA
STEPHEN CHOU, Massachusetts Institute of Technology, USA
DAVID LUGATO, French Alternative Energies and Atomic Energy Commission, France
SAMAN AMARASINGHE, Massachusetts Institute of Technology, USA

Tensor algebra is a powerful tool with applications in machine learning, data analytics, engineering and the physical sciences. Tensors are often sparse and compound operations must frequently be computed in a single kernel for performance and to save memory. Programmers are left to write kernels for every operation of interest, with different mixes of dense and sparse tensors in different formats. The combinations are infinite, which makes it impossible to manually implement and optimize them all. This paper introduces the first compiler technique to automatically generate kernels for any compound tensor algebra operation on dense and sparse tensors. The technique is implemented in a C++ library called `taco`. Its performance is competitive with best-in-class hand-optimized kernels in popular libraries, while supporting far more tensor operations.

CCS Concepts: • **Software and its engineering → Source code generation**; **Domain specific languages**; *Software libraries and repositories*; • **Mathematics of computing** → *Mathematical software performance*;

Additional Key Words and Phrases: tensors, tensor algebra, linear algebra, sparse data structures, performance, parallelism, code generation, merge lattices, iteration graphs

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

1 INTRODUCTION

Tensor algebra is a powerful tool for computing on multidimensional data and has many applications in machine learning, data analytics, engineering, and science [Abadi et al. 2016; Anandkumar et al. 2014; Bader et al. 2008; Einstein 1916; Feynman et al. 1963; Kolecki 2002]. Tensors generalize vectors and matrices to any number of dimensions and permit multilinear computations. Tensors of interest are often sparse, which means they mostly contain zeros that can be compressed away. For example, large real-world datasets used in data analytics, such as Netflix Ratings [Bennett et al. 2007], Facebook Activities [Viswanath et al. 2009], and Amazon Reviews [McAuley and Leskovec 2013], are conveniently cast as sparse tensors. The Amazon Reviews tensor, in particular, contains 1.5×10^{19} components corresponding to 107 exabytes of data (assuming 8 bytes are used per component), but only 1.7×10^{9} of the components (13 gigabytes) are non-zero.

Authors' addresses: F. Kjolstad, 32 Vassar St, 32-G778, Cambridge, MA 02139; email: fred@csail.mit.edu; S. Kamil, 104 Fifth Ave, 4th Floor, New York, NY 10011; email: kamil@adobe.com; S. Chou, 32 Vassar St, 32-G778, Cambridge, MA 02139; email: s3chou@csail.mit.edu; D. Lugato, CEA-CESTA, 33114 Le Barp, France; email: david.lugato@cea.fr; S. Amarasinghe, 32 Vassar St, 32-G744, Cambridge, MA 02139; email: saman@csail.mit.edu.

2475-1421/2017/10-ART77
https://doi.org/10.1145/3133901

Optimized kernels that compute compound tensor operations can be very complex. First, sparse input tensors are compressed into indexed data structures that the kernels must operate on. Secondly, only non-zero output values should be produced, and the calculations differ between computed components depending on the input tensors that contribute non-zero values. Finally, sparse tensor data structures typically do not support constant-time random access, so kernels must carefully orchestrate co-iteration over multiple tensors.

The current approach is to manually write high-performance code for tensor operations. Libraries provide a limited set of hand-optimized operations and programmers compute compound operations through a sequence of supported operations using temporary tensors. This reduces locality and efficiency, and for some compound operations the temporaries are much larger than the kernel's inputs and output. On the other hand, it is infeasible to write optimized code for every compound operation by hand because of the combinatorial explosion of all possible compound operations, tensor orders, and tensor formats. A compiler approach is therefore needed to generate fast kernels from a high-level notation such as the ubiquitous tensor index notation for tensor expressions [Ricci-Curbastro and Levi-Civita 1901].

Prior to this work, there existed no general mechanism that can generate code for compound tensor operations with sparse tensors. To the best of our knowledge, we present the first compiler technique that can generate kernels for all sparse and dense tensor index notation expressions. This includes dense and sparse linear algebra expressions. Our technique generates code entirely from tensor index notation and simple format descriptors that fully specify the compressed data structures. Thus, we do not perform pointer alias or dependency analysis at compile time, nor at runtime as in inspector-executor systems. The technique can be used in libraries such as TensorFlow [Abadi et al. 2016] and Eigen [Guennebaud et al. 2010], or with languages such as MATLAB [MATLAB 2014], Julia [Bezanson et al. 2012] and Simit [Kjolstad et al. 2016]. The contributions of this paper are:

tensor storage formats that separately designate each dimension as dense or sparse and specify the order in which dimensions are stored, which can describe several widely used formats but generalizes to many more (Section 3);

iteration graphs that describe how to co-iterate through the multi-level index data structures of the sparse operands in a compound tensor expression (Section 4);

merge lattices that describe how to merge the index data structures of sparse tensors that are used in the same tensor expression (Section 5); and a

code generation algorithm that uses the above concepts to generate efficient code that computes a tensor expression with dense, sparse, and mixed operands (Section 6).

To demonstrate our approach we have developed a C++ library called `taco`, short for Tensor Algebra COmpiler (Section 7).[1] We compare `taco`-generated code to hand-written implementations from state-of-the-art widely used libraries. The comparison shows that `taco` generates efficient code for simpler kernels such as sparse matrix-vector multiplication as well as complex kernels like the matricized tensor times Khatri-Rao product [Bader and Kolda 2007] (Section 8).

2 EXAMPLE

Tensors generalize matrices to any number of dimensions (also often called modes, though we will use 'dimensions' in the rest of this paper). The number of dimensions that a tensor has is its order. Thus, scalars are 0th-order tensors, vectors are 1st-order tensors, and matrices are 2nd-order tensors. Consider a simple 3rd-order tensor kernel, the tensor-times-vector multiplication (TTV):

$$A_{ij} = \sum_k B_{ijk} c_k$$

[1] The `taco` library and tools are available under the MIT license at http://tensor-compiler.org.

```
1  for (int i = 0; i < m; i++) {
2
3
4
5    for (int j = 0; j < n; j++) {
6      int pB2 = i * n + j;
7      int pA2 = i * n + j;
8
9
10     for (int k = 0; k < p; k++) {
11       int pB3 = pB2 * p + k;
12
13
14
15
16
17
18       A[pA2] += B[pB3] * c[k];
19
20
21
22     }
23   }
24 }
```

Fig. 1. $A_{ij} = \sum_k B_{ijk} c_k$

```
1  for (int pB1 = B1_pos[0];
2           pB1 < B1_pos[1];
3           pB1++) {
4    int i = B1_idx[pB1];
5    for (int pB2 = B2_pos[pB1];
6             pB2 < B2_pos[pB1+1];
7             pB2++) {
8      int j = B2_idx[pB2];
9      int pA2 = i * n + j;
10     for (int pB3 = B3.pos[pB2];
11              pB3 < B3.pos[pB2+1];
12              pB3++) {
13       int k = B3_idx[pB3];
14
15
16
17
18       A[pA2] += B[pB3] * c[k];
19
20
21
22     }
23   }
24 }
```

Fig. 2. $A_{ij} = \sum_k B_{ijk} c_k$ (sparse B)

```
1  for (int pB1 = B1_pos[0];
2           pB1 < B1_pos[1];
3           pB1++) {
4    int i = B1_idx[pB1];
5    for (int pB2 = B2_pos[pB1];
6             pB2 < B2_pos[pB1+1];
7             pB2++) {
8      int j = B2_idx[pB2];
9      int pA2 = i * n + j;
10     int pB3 = B3_pos[pB2];
11     int pc1 = c1_pos[0];
12     while (pB3 < B3_pos[pB2+1] &&
13            pc1 < c1_pos[1]) {
14       int kB = B3_idx[pB3];
15       int kc = c1_idx[pc1];
16       int k = min(kB, kc);
17       if (kB == k && kc == k) {
18         A[pA2] += B[pB3] * c[pc1];
19       }
20       if (kB == k) pB3++;
21       if (kc == k) pc1++;
22     }
23   }
24 }
```

Fig. 3. $A_{ij} = \sum_k B_{ijk} c_k$ (sparse B, c)

This notation, which we use both in our presentation and as the input to our technique, is a variant of tensor index notation developed by Ricci-Curbastro and Levi-Civita [1901]. The notation uses index variables to relate each component in the result to the components in the operands that are combined to produce its value. In this example, every component A_{ij} of the output is the inner product of a vector from the last dimension of B with c.

The kernel that evaluates this expression depends entirely on the formats of the three tensors. The simplest case is when all of the tensors are dense, and the kernel for this is shown in Fig. 1. The loops iterate over the $m \times n \times p$ rectangular iteration space of the tensor dimensions and, since the tensors are dense, the physical location of every component can be computed.

If most components in B are zero, then it is more efficient to only store the non-zeros. This can be done with a sparse tensor data structure such as compressed sparse fiber (CSF) where every dimension is compressed [Smith and Karypis 2015]. However, the kernel must be written to traverse the data structure to find non-zeros. Fig. 2 shows the code for computing the expression when B is stored in CSF. The loops iterate over the subset of each dimension of B that contains non-zeros (lines 1–13), performs a multiplication, and stores the result in the correct place in A (line 18).

If most components in c are also zero, then it can be compressed as well. However, the kernel must now simultaneously iterate over and merge the dimension indexed by k in B and c. This means the kernel only computes a value if both B and c have non-zeros at a location. The merge code is shown in Fig. 3 on lines 10–22. A `while` loop iterates while both B and c have values remaining and only computes if both have a value at a location.

These examples show that different kernels for the same expression look quite different depending on the formats of the tensors. Moreover, these are only three out of the 384 possible kernels for all the combinations of formats our technique supports for this expression alone. Implementing each by hand is not realistic. In addition, the number of possible compound expressions is unbounded. Some can be efficiently computed with multiple simpler kernels strung together, while others are best evaluated as a single compound kernel. The technique presented in this paper leaves the decision to the user, but obviates the need to write the kernels by hand. This makes it possible to mix and match formats and automatically generate kernels for any tensor algebra operation.

There are two primary reasons for sparse tensor algebra kernel complexity. First, sparse data structures can only be accessed efficiently in one direction. For example, the indices of B follow the

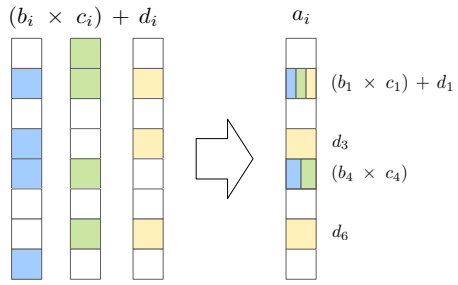


Fig. 4. A sparse $a_i = b_i c_i + d_i$ computation. The colors in the output show the vectors that are combined to produce its values and the expressions show the calculation for each non-zero component in the output.

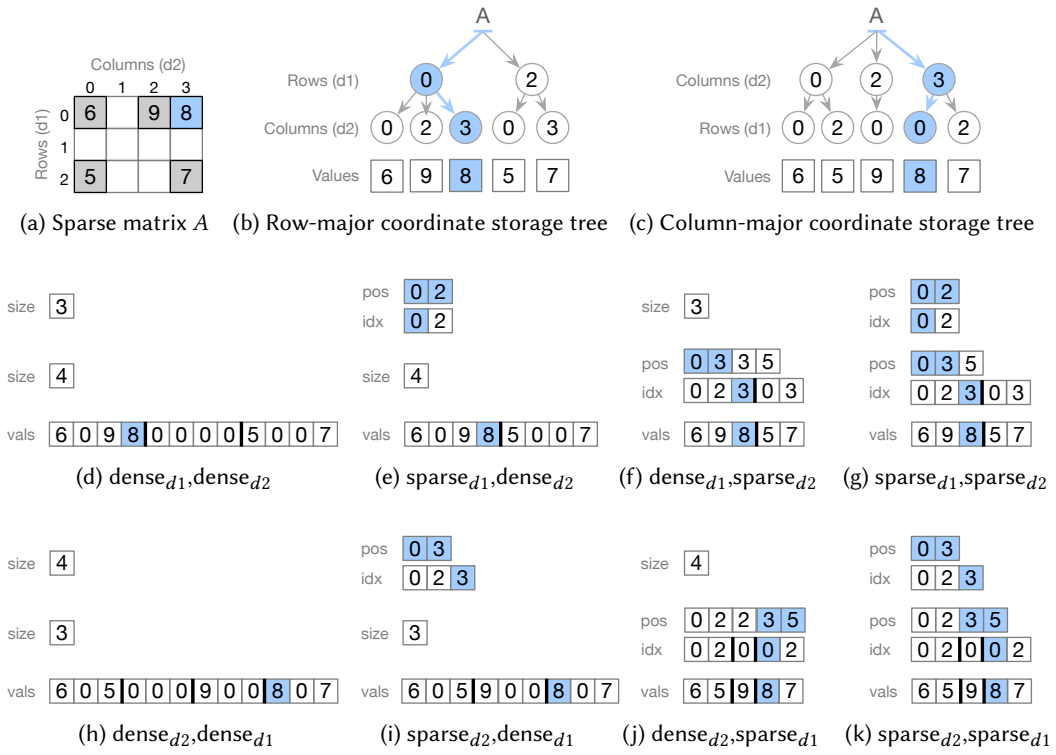
order of the dimensions. It is sometimes preferable to choose a different ordering, such as storing B's last dimension first. This is common in sparse linear algebra, where the compressed sparse row (CSR) format goes from rows to columns while the compressed sparse column (CSC) format goes from columns to rows. However, any ordering of dimensions must be respected by the ordering of the kernel loop nests. We address loop ordering with a concept called iteration graphs in Section 4.

The second reason for sparse kernel complexity is the merging of sparse tensor indices. We saw this when B and c were both sparse, but consider the expression $a_i = b_i c_i + d_i$, where the component-wise product of two vectors is added to a third. The merge with d is different from the bc merge because the operation is an addition, which produces a value if *either* operand is non-zero. (Multiplication produces a value if *both* operands are non-zero.) Thus, the three-way merge of b, c and d produces only a value if either both b and c have non-zeros or if d has a non-zero. Fig. 4 shows an example. When only b or c has a non-zero value, but not d, no output is produced. When d has a non-zero value an output is always produced; however, it is added to the product of b and c if both have non-zero values. Finally, an output is also produced if b and c have non-zero values but not d. For efficiency, only the non-zero values in the inputs are visited and only the non-zero values in the output are produced. As shown in the figure, each non-zero output is calculated using a simplified expression containing only non-zero inputs, which avoids unnecessary operations. We address the complexity of merging with a new concept we call merge lattices in Section 5.

3 TENSOR STORAGE

Tensors can have any order (dimensionality) and it is therefore not tractable to enumerate formats for all tensors. To support tensors of any order, it is necessary to construct formats from a bounded number of primitives. This section describes a way to define storage formats for tensors of any order by specifying whether each dimension is dense or sparse and declaring the order in which the dimensions should be stored. For sparse levels we use the same compression techniques as used in the CSR format. However, we allow these levels to be combined to construct sparse storage for any tensor. Since each level can independently be dense or sparse, users can build custom formats for specific tensors. A second benefit of describing each dimension separately is that it leads to a composable code generation algorithm that supports tensors of any order.

Consider the 2nd-order tensor (matrix) A shown in Fig. 5a. The simplest storage format is a dense multidimensional array, but it is wasteful if most components are zeros. Intuitively, we find it convenient to think of a tensor as a tree with one tree level per dimension (plus a root), as shown for matrix A in Figs. 5b–c. In this formulation, each tree-path from the root to a leaf represents a tensor coordinate with a non-zero value. The non-root nodes along the path are the coordinates and the non-zero value is attached to the leaf node. Finally, depending on the order in which the



(a) Sparse matrix A (b) Row-major coordinate storage tree (c) Column-major coordinate storage tree

(d) dense$_{d1}$,dense$_{d2}$ (e) sparse$_{d1}$,dense$_{d2}$ (f) dense$_{d1}$,sparse$_{d2}$ (g) sparse$_{d1}$,sparse$_{d2}$

(h) dense$_{d2}$,dense$_{d1}$ (i) sparse$_{d2}$,dense$_{d1}$ (j) dense$_{d2}$,sparse$_{d1}$ (k) sparse$_{d2}$,sparse$_{d1}$

Fig. 5. A sparse matrix (2nd-order tensor) is shown in (a), with trees that describe its non-zeros, ordered from rows to columns in (b) and from columns to rows in (c). Figures (d)–(g) show the row-major matrix formats we support, while figures (h)–(k) show the column-major formats. Similar formats apply to tensors of other orders. See Section 8.5 for use cases of these formats.

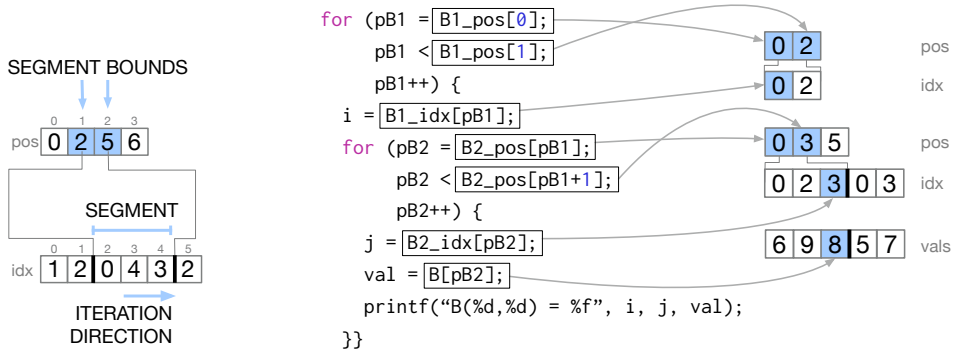
dimensions of A are stored, the levels of the tree can occur in different order, e.g. (d_1, d_2) or (d_2, d_1) where d_i denotes a dimension of the matrix.

In our technique, tensor values are always stored in a separate array, and the tensor format index arrays are used to determine their location in the array. For each kind of level storage, we store index metadata associated with the corresponding dimension:

dense requires only storing the size of the dimension, since it stores values—both zeros and non-zeros—for every coordinate in the dimension.

sparse stores only the subset of the corresponding dimension that has non-zero values. This requires two index arrays, `pos` and `idx`, that together form a segmented vector with one segment per entry in the previous dimension (parent node in the tree). The `idx` array stores all the non-zero indices in the dimension, while the `pos` array stores the location in the `idx` array where each segment begins. Thus segment i is stored in locations `pos[i]:pos[i+1]` in the `idx` array (we add a sentinel at the end of the `pos` array with the size of the `idx` array). We store each segment in `idx` in sorted order.

Note that the index arrays in a sparse dimension are the same as those in the CSR matrix format. In addition to higher-order tensors, our formulation can represent several common sparse matrix formats. Figs. 5d–k shows all eight ways to store a 2nd-order tensor using our technique. The first column shows dense row- and column-major storage. The second column shows the



(a) Iteration through sparse level (b) Iteration through two sparse levels

Fig. 6. (a) Iteration over a sparse storage level. Segment bounds are stored in pos, while idx stores index values for the segments. (b) Iteration through a (sparse$_{d1}$,sparse$_{d2}$) matrix, showing the correspondence between code and storage. The arrows show the state of loop variables when printing the highlighted value.

(sparse$_{d1}$,dense$_{d2}$) format and its column-first equivalent, neither of which is commonly used but can be useful in some circumstances (see Section 8.5). The third column shows the CSR and CSC formats. Both are represented as (dense,sparse), but the order of dimensions is switched. Finally, the fourth column shows the (sparse,sparse) formats, which are equivalent to doubly compressed sparse row (DCSR) and the corresponding column-major format DCSC [Buluc and Gilbert 2008]. The number of formats increases exponentially as tensor dimensionality increases (by $d!2^d$ in fact, where d is the order of the tensor), making hand-coding intractable. Other important sparse formats this technique supports include sparse vectors and the CSF format for higher-order tensors, which correspond to formats that use a sparse level for every tensor dimension.

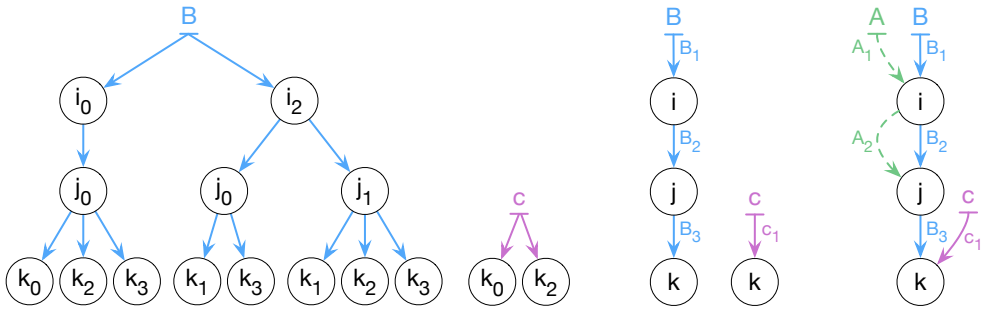
A sparse storage dimension does not allow efficient random access to indices and values, but is optimized for iteration in a specific order. Fig. 6a shows how to iterate over a sparse level. The pos array gives the bounds for each segment in the idx array, and code iterates over indices in a segment with unit stride. Fig. 6b shows the correspondence between code for iterating through a (sparse$_{d1}$,sparse$_{d2}$) 2nd-order tensor, and the tensor's storage.

The two kinds of level storage we support express a large space of tensor formats and operations, including blocked sparse matrix-vector multiplication, which we cast as the multiplication of a 4th-order tensor (a blocked matrix with two inner dense dimensions storing small dense blocks of non-zeros) and a 2nd-order tensor (a blocked vector). Describing the space of tensor storage formats in this manner enables us to support an unbounded number of formats, gives us support for blocked linear algebra for free, and makes code generation modular for each storage level (Section 6).

4 ITERATION GRAPHS

Iteration graphs are a compiler intermediate representation that describe how to iterate over the non-zero values of a tensor expression. This requires generating code to simultaneously iterate over the tensor storage trees of the expression's operands. Iteration graphs are constructed from tensor index expressions and are used to generate such code. They are general and can represent any tensor expression from simple sparse matrix-vector multiplications (SpMV) to complex compound expressions such as the matricized tensor times Khatri-Rao product (MTTKRP), as we will show.

Definition 4.1. An iteration graph is a directed graph $G = (V, P)$ with a set of index variables $V = \{v_1, ..., v_n\}$ and a set of tensor paths $P = \{p_1, ..., p_m\}$. A tensor path is a tuple of index variables.



(a) Example coordinate trees for 3rd-order tensor B and vector c (b) Iteration graphs of B_{ijk} and c_k (c) $A_{ij} = \sum_k B_{ijk} c_k$

Fig. 7. Figure (a) shows example tensor storage trees for a 3rd-order tensor B and a vector c. Any code to compute with these tensors must somehow traverse these trees. Figure (b) shows the iteration graphs of the tensor access expressions B_{ijk} and c_k. A path through an iteration graph is a symbolic summary of all the paths from the root to leaves in a tensor storage tree. Figure (c) shows the iteration graph for a tensor-times-vector multiplication, with one path for each tensor access sub-expression.

As described in the previous section, tensors can be represented by trees where each level corresponds to a dimension that can be either dense or sparse. Fig. 7a shows two examples, a 3rd-order tensor B and a vector c, while Fig. 7b shows iteration graphs for the tensor access expressions B_{ijk} and c_k. Each node in the iteration graphs corresponds to a loop in a loop nest. Each of these iteration graphs contains a path through the index variables in the access expressions. We call these *tensor paths* and they symbolically summarize all traversals from root to leaves in B and c. The purpose of iteration graphs is to reason about these traversals. Fig. 7c shows the iteration graph for a tensor-times-vector multiplication. It has one tensor path per tensor access sub-expression and represents the simultaneous iteration over the tensor storage trees of both B and c. It also shows the tensor path for the output A. In general, iteration graphs can have any number of tensor paths.

Tensor path edges that lead into the same node represent tensors that need to be merged at this dimension. That is, to simultaneously iterate over multiple tensor storage trees, they have to be merged. In the tensor-times-vector multiplication example, this means merging the last dimension of B with c when iterating over the k index variable. Since it is a multiplication the merge is a conjunction (and), as both values must be non-zero for the result to be non-zero. If it was an addition, then the merge would be a disjunction (or). If more than two tensor dimensions were to be merged, then the merge would be a combination of conjunctions and disjunctions, as Section 5 will discuss in further detail.

Fig. 8 shows eight more iteration graph examples for kernels ranging from a simple SpMV in Fig. 8a to MTTKRP in Fig. 8h. All of these examples require merging if all the operands are sparse. Note that the blocked sparse matrix-vector multiplication in Fig. 8f is cast as a 4th-order tensor times 2nd-order tensor multiplication. Finally, note the sampled dense-dense matrix product (SDDMM) kernel in Fig. 8d, where B is sparse. This is a kernel from machine learning [Zhao 2014], and since B is sparse the number of inner products to evaluate in CD can be decreased from $\Theta(n^2)$ to $\Theta(\text{nnz}(B))$ by evaluating the whole expression in a single loop nest.

Iteration graphs are constructed from an index expression by creating one tensor path for every tensor access expression. The tensor paths are ordered based on the order of the index variables in the access expression and the order of the dimensions in the tensor storage tree. So B_{ij} becomes the path $i \rightarrow j$ if the first dimension is the first level of the storage tree (e.g. CSR) and $j \rightarrow i$ otherwise

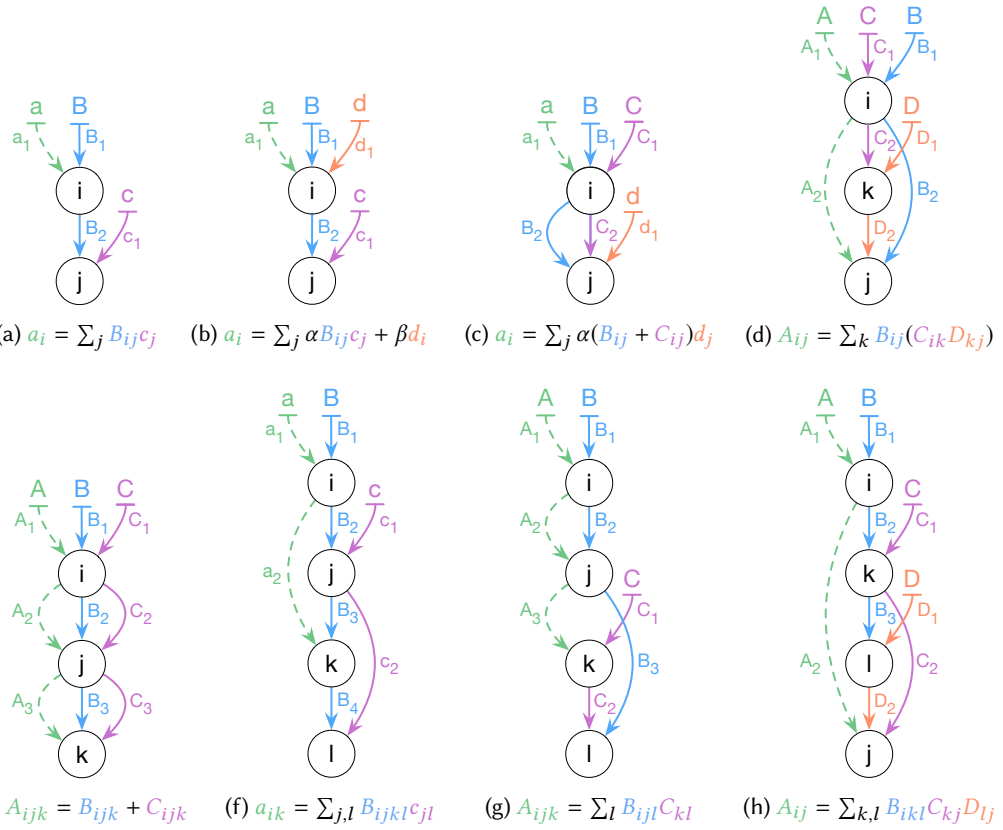


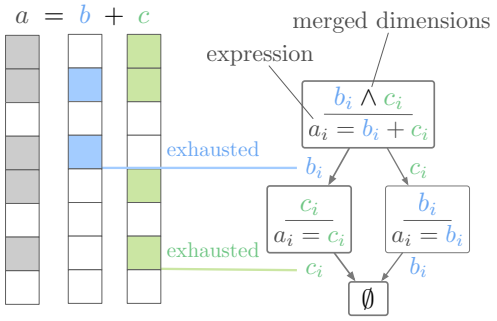
Fig. 8. Iteration graphs for (a) matrix-vector multiplication, (b) scaled matrix-vector multiplication plus a scaled vector, (c) scaled sum of two matrices times vector, (d) sampled dense-dense matrix product, (e) tensor addition, (f) *blocked* matrix-vector multiplication, (g) tensor-times-matrix multiplication (TTM), and (h) matricized tensor times Khatri-Rao product.

(e.g. CSC). Finally, the iteration graph's index variables are ordered into a forest such that every tensor path edge moves from index variables higher up to index variables lower down. This is necessary to ensure the tensor trees are traversed from roots to leaves.

Iteration graphs can be constructed for any tensor index expression, but the code generation technique described in Section 6 does not support iteration graphs with cycles. The reason is that back edges require traversal in an unnatural direction for a format, such as traversing a CSR matrix from columns to rows. This would require code that scans indices to find components and is outside the scope of this paper. Cycles occur when an expression operates on tensors with incompatible formats, such as adding a CSR matrix to a CSC matrix or transposing a matrix. The solution is to provide a function to change tensor formats that can be used to break cycles and transpose tensors.

5 MERGE LATTICES

Index variables that access dimensions of more than one tensor must iterate over the merged indices of those dimensions. The type of merge depends on the tensor index expression. If the tensors are multiplied then the merge is a conjunction ($\wedge$), because both factors must be non-zero for the result to be non-zero ($a \cdot 0 = 0$). This means the merged iteration space is the intersection of the iteration




```
int pa1 = a1_pos[0];
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];

// one loop per lattice point
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
  int ib = b1_idx[pb1];
  int ic = c1_idx[pc1];
  int i = min(ib, ic);

  // one case per lattice point below loop lattice point
  if (ib == i && ic == i)
    a[pa1++] = b[pb1] + c[pc1];
  else if (ib == i)
    a[pa1++] = b[pb1];
  else if (ic == i)
    a[pa1++] = c[pc1];

  if (ib == i) pb1++;
  if (ic == i) pc1++;
}
while (pb1 < b1_pos[1]) {
  a[pa1++] = b[pb1++];
}
while (pc1 < c1_pos[1]) {
  a[pa1++] = c[pc1++];
}
```

Fig. 9. Sparse vector addition ($a_i = b_i + c_i$) example. The middle shows the merge lattice for the expression, where the disjunctive merge becomes three conjunctive lattice points ordered on b and c running out of values. The right shows C code generated from the merge lattice.

spaces of the merged dimensions. Conversely, if the two tensors are added then the merge is a disjunction ($\vee$), since only one term needs to be non-zero for the result to be non-zero ($a + 0 = a$). This corresponds to the union of the iteration spaces of the merged dimensions. If more than two dimensions are merged then the merge is a combination of conjunctions and disjunctions that mirror the index expression. For example, $(b_i + c_i)d_i \implies (b_i \vee c_i) \wedge d_i$.

Merge lattices are motivated by the cost of a disjunctive merge, where every loop iteration must check that each merged index has more values left to avoid out-of-bounds accesses. The two-way merge algorithm is used to avoid these expensive checks [Knuth 1973, Chapter 5.2.4]. It has three loops: one that iterates until either of the indices is exhausted (runs out of values) and two more that process the rest of the unexhausted index. We generalize this insight with merge lattices that are used to generate efficient loops for any merge expression.

Consider a concrete example, sparse vector addition, which requires a disjunctive merge. Fig. 9 (left) shows an example with a result a and two operands b and c, of which c has more values. A two-way merge iterates over both operands until either is exhausted. At each step, $b + c$, b, or c is added to a depending on whether both b and c, just b, or just c have values at that coordinate. After one of the vectors is exhausted, the two-way merge iterates over the remaining entries in the other vector. In this example b is exhausted first, so the remaining values of c are added to a.

Fig. 9 (center) shows the merge lattice for sparse vector addition, with one lattice point for each of three cases: where b and c have values, where only b has values, or where only c has values. Each lattice point contains the dimensions that contain values for its case ($b_i \wedge c_i$, b_i, or c_i) and the sub-expression to compute. A lattice arrow corresponds to when a dimension has been exhausted.

A merge lattice can merge any number of dimensions and can therefore have any number of lattice points. Lattice points serve two purposes. First, they describe the necessary loops to merge the dimensions. The top lattice point describes the loop where every dimension still has values left. When any dimension is exhausted, we move along an arrow to another lattice point that describes

the iteration over the remaining dimensions. This process repeats until we reach bottom, where no more values remain to be merged. The second purpose is to describe the cases that must be considered within each loop. The iterations described by any lattice point may encounter different cases depending on which subset of the merged dimensions have values at a given location. The necessary cases are described by the lattice points dominated by a given lattice point, including itself, and each lattice point contains the expression to compute for its case.

Definition 5.1. A merge lattice L is a lattice comprising n lattice points $\{L_1, ..., L_n\}$ and a meet operator. Each lattice point L_p has associated with it a set of tensor dimensions $T_p = \{t_{p1}, ..., t_{pk}\}$ to be merged conjunctively (i.e. $t_{p1} \wedge ... \wedge t_{pk}$) and an expression $expr_p$ to be evaluated. The meet of two lattice points L_1 and L_2 with associated tensor dimensions T_1 and T_2 respectively is a lattice point with tensor dimensions $T_1 \cup T_2$. We say $L_1 \leq L_2$ if and only if $T_1 \subseteq T_2$, in other words if L_2 has tensor dimensions that are exhausted in L_1 but not vice versa.

Fig. 9 (right) shows code generated from the merge lattice. Each lattice point results in a while loop: lattice point $\boxed{b_i \wedge c_i}$ in the loop on lines 6–21, $\boxed{b_i}$ in the loop on lines 22–24, and $\boxed{c_i}$ in the loop on lines 25–27. Furthermore, $\boxed{b_i \wedge c_i}$ dominates three non-bottom lattice points and the resulting loop therefore considers three cases on lines 12–17. Each case compares the index variable for each of the tensor dimensions in the case to the merged index variable for the loop iteration, which is the smallest of the dimension index variables and is computed on line 9. Finally, the dimension position variables are incremented as needed on lines 19-20.

5.1 Construction

In this section we describe an algorithm to construct a merge lattice for an index variable in an index expression. The algorithm traverses the index expression tree from the bottom up, constructing merge lattices at the leaves and successively combining merge lattices at the internal nodes using the following operators:

Definition 5.2. Let the conjunction of two lattice points L_p and L_q with the operator op be a new lattice point with associated tensor dimensions $T_p \cup T_q$ and expression $expr_p \ op \ expr_q$.

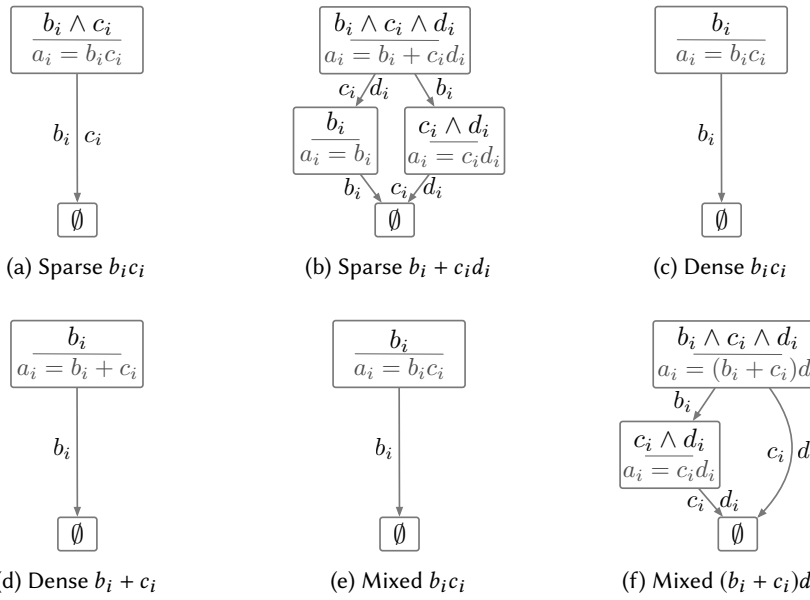
Definition 5.3. Let the conjunctive merge of two lattices L^1 and L^2 with the operator op, denoted $L^1 \wedge_{op} L^2$, be a new lattice with lattice points constructed by applying the conjunction operation defined above to every pair of lattice points in the Cartesian product $(L^1_0, \ldots, L^1_n) \times (L^2_0, \ldots, L^2_m)$.

Definition 5.4. Let the disjunctive merge of two lattices L^1 or L^2 with the operator op, denoted $L^1 \vee_{op} L^2$, be a new lattice containing all lattice points in $L^1 \wedge_{op} L^2$, L^1, and L^2.

The following algorithm constructs a merge lattice from an index expression and an index variable i. The algorithm works bottom-up on the index expression and, for each sub-expression, applies one of the following rules:

- *Tensor Access:* construct a merge lattice with one lattice point that contains the dimension indexed by i in the access expression. The lattice point expression is the tensor access. If no dimension is accessed by i, then construct an empty lattice point.
- *Conjunctive Operator (e.g. multiply):* compute and return the conjunctive merge of the operand merge lattices with the operator.
- *Disjunctive Operator (e.g. add):* compute and return the disjunctive merge of the operand merge lattices with the operator.

To build an intuition consider the expression $a_i = b_i + c_i d_i$, a combined addition and component-wise multiplication. The algorithm first creates lattices from the leaves with the tensor access rule.



(a) Sparse $b_i c_i$ (b) Sparse $b_i + c_i d_i$ (c) Dense $b_i c_i$

(d) Dense $b_i + c_i$ (e) Mixed $b_i c_i$ (f) Mixed $(b_i + c_i) d_i$

Fig. 10. Merge lattices for several expressions. (a)–(b) have only sparse operands, (c)–(d) have only dense operands, and (e)–(f) have mixed operands where c is dense while b and d are sparse. Lattices with dense operands are optimized as described in Section 5.2.

It then creates a merge lattice for $c_i d_i$ by computing the conjunctive merge of the operand merge lattices. This merge lattice has one lattice point ($\boxed{c_i \wedge d_i}$). In the inline notation parentheses enclose lattices and boxes lattice points, which are ordered so that every point appears after its ancestors. Finally, since the top-level expression is an addition, the algorithm computes the disjunctive merge ($\boxed{b_i}$) $\vee_+$ ($\boxed{c_i \wedge d_i}$) = ($\boxed{b_i \wedge c_i \wedge d_i}$, $\boxed{b_i}$, $\boxed{c_i \wedge d_i}$). The final merge lattice is shown in Fig. 10b. The top results in a loop over all three dimensions. If either c_i or d_i is exhausted, the lattice loop exits and the loop for $\boxed{b_i}$ takes over. Otherwise, if b_i is exhausted first, the loop for $\boxed{c_i \wedge d_i}$ takes over.

5.2 Optimizations

Merge lattices constructed with the algorithm in Section 5.1 merge all accessed dimensions. This is necessary if they are sparse, but if some are dense then the lattice can be optimized in three ways:

Dense iteration spaces are the same, so if more than one of the merged dimensions is dense then we only need to iterate over one of them. Fig. 10c and Fig. 10d show merged dense dimensions in a vector product and in a vector addition respectively.

Dense iteration spaces are supersets of sparse iteration spaces, so when a dense dimension is exhausted we are done. This means we can remove lattice points below the top point that do not merge every dense dimension, because if a dense dimension is exhausted in the top lattice point then there are no more values to merge. Fig. 10d shows a dense vector addition where every lattice point except one is removed.

Dense iteration spaces support random access, so when conjunctively merged with sparse iteration spaces we can iterate over the sparse iteration space and pick values from the dense. Fig. 10e shows a vector product lattice with sparse b and dense c. The lattice iterates over b because c is a superset with random access. This optimization is used in the SpMV kernel.

6 CODE GENERATION

In this section, we describe how tensor storage descriptors, iteration graphs, and merge lattices are used in a recursive algorithm to produce kernels from compound index expressions. The algorithm produces loops that iterate over the expression's merged iteration space (Section 6.1). It also populates these loops with statements that compute result values (Section 6.2) and assemble result indices (Section 6.3). These statements may be combined into a single kernel that concurrently assembles indices and computes values, or separated into one kernel that only performs assembly and another that only computes. The latter is useful when values are recomputed while the non-zero structure remains the same. Finally, we will discuss parallel code generation (Section 6.4).

6.1 Code Generation Algorithm

Pseudo-code for the recursive code generation algorithm is given in Fig. 11a. Compiler code is colored blue and emitted code is black and red in quotation marks. Blue text inside quotation marks emits the value of a compiler variable, such as the name of an index variable. Fig. 11b shows the generated code for 256×256 sparse matrix addition, where A is dense and the inputs B and C are CSR matrices. The code assumes the values of the dense output matrix have been pre-initialized to zero. Numbered labels relate the code generation algorithm to the emitted code.

The code generation algorithm is a recursive function. It is called with an index expression and the first index variable from the expression's iteration graph, and recurses on the index variables in the iteration graph in the forest ordering. Fig. 11c shows the iteration graph for matrix addition. At each recursive level, the code generation algorithm first constructs a merge lattice from the index variable and index expression. Figs. 11d and 11e show the merge lattices created for i and j in the sparse matrix addition example. Next, it initializes sparse `idx` variables (1) followed by a loop that emits one `while` loop per lattice point in level order (2). Each `while` loop starts by loading (3) and merging (4) sparse `idx` variables. The resulting merged `idx` variable is the coordinate in the current index variable's dimension. Dense `pos` variables (e.g., `pB1`) are then computed by adding the merged `idx` variable to the start of the variable's corresponding tensor slice (5).

Next, the algorithm emits one `if-else if` case per lattice point in the sub-lattice dominated by the loop lattice point (7). For example, the sub-lattice of the top lattice point is the whole lattice (a sub-lattice includes itself). Sub-lattices of other lattice points include all points reachable from them. Inside the lattice point's `if-else if` case, the `code-gen` function recursively calls itself to generate code for each child of the current index variable in the iteration graph forest ordering. Next, it emits code to insert index entries and to compute values as necessary at this loop level; this is described in more details in Sections 6.2 and 6.3. Finally, the algorithm emits code to conditionally increment sparse `pos` variables if they took part in the current merged coordinate (8).

In the algorithm we have described, the `pos` variables are named based on the tensors they access. However, in the presence of repeated accesses of the same tensor, such as in the expression $A_{ij} = \sum_k B_{ik} B_{kj}$, a separate mechanism ensures that unique names are associated with the `pos` variables related to each tensor access.

6.2 Compute Code

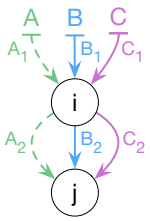
We have shown how to generate loops that iterate over input tensors, but so far we have left out what statements to insert into those loop nests to compute result values. The code generation algorithm in Fig. 11 calls three functions to emit compute code, namely `emit-available-expressions` (6), `emit-reduction-compute` (7), and `emit-compute` (7). For any given index variable only one of these functions emits code, depending on whether it is the last free variable in the recursion, above the last free variable, or below it.

```
code-gen(index-expr, iv)  # iv is the index variable
  let L = merge-lattice(index-expr, iv)

  # initialize sparse pos variables
  for Dj in sparse-dimensions(L)
    emit "int pDj = Dj_pos[pDj-1];"

  for Lp in L
    # while all merged dimensions have more values
    emit "while(until-any-exhausted(merged-dimensions(Lp))) {"

    # initialize sparse idx variables
    for Dj in sparse-dimensions(Lp)
      emit "int ivDj = Dj_idx[pDj];"

    # merge sparse idx variables
    emit   "int iv = min(["ivDj," Dj in sparse-dimensions(Lp)]);"

    # compute dense pos variables
    for Dj in dense-dimensions(Lp)
      emit "int pDj = (pDj-1 * Dj_size) + iv;"

    # compute expressions available at this loop level
    emit-available-expressions(index-expr, iv)    # Section 6.2

    # one case per lattice point below Lp
    for Lq in sub-lattice(Lp)
      emit "if (equals-iv(["ivDj" Dj in sparse-dimensions(Lq)])) {"
        for child-iv in children-in-iteraton-graph(iv)
          code-gen(expression(Lq), child-iv)
        emit-reduction-compute()  # Section 6.2
        emit-index-assembly()     # Section 6.3
        emit-compute()            # Section 6.2
        if result dimension Dj is accessed with iv
          emit "pDj++;"
      emit "}"

    # conditionally increment the sparse pos variables
    for Dj in sparse-dimensions(Lp)
      emit "if (ivDj == iv) pDj++;"
    emit "}"
```

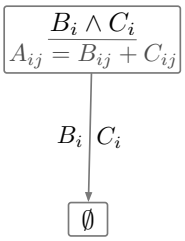
(a) Recursive algorithm to generate code for tensor expressions

```
for (int i = 0; i < B1_size, i++) {
  int pB1 = (0 * B1_size) + i;
  int pC1 = (0 * C1_size) + i;
  int pA1 = (0 * A1_size) + i;

  int pB2 = B2_pos[pB1];
  int pC2 = C2_pos[pC1];
  while (pB2 < B2_pos[pB1+1] &&
         pC2 < C2_pos[pC1+1]) {
    int jB = B2_idx[pB2];
    int jC = C2_idx[pC2];
    int j = min(jB, jC);
    int pA2 = (pA1 * A2_size) + j;

    if (jB == j && jC == j)
      A[pA2] = B[pB2] + C[pC2];
    else if (jB == j)
      A[pA2] = B[pB2];
    else if (jC == j)
      A[pA2] = C[pC2];

    if (jB == j) pB2++;
    if (jC == j) pC2++;
  }

  while (pB2 < B2_pos[pB1+1]) {
    int j = B2_idx[pB2];
    int pA2 = (pA1 * A2_size) + j;
    A[pA2] = B[pB2];
    pB2++;
  }

  while (pC2 < C2_pos[pC1+1]) {
    int j = C2_idx[pC2];
    int pA2 = (pA1 * A2_size) + j;
    A[pA2] = C[pC2];
    pC2++;
  }
}
```

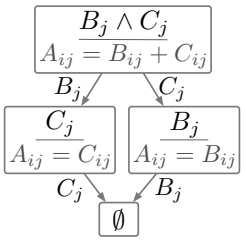
(b) Generated sparse matrix addition code



(c) Iteration graph for matrix addition



(d) Dense merge lattice for i



(e) Sparse merge lattice for j

Fig. 11. (a) Recursive code generation algorithm for tensor index notation. (b) Generated code for a 256×256 sparse matrix addition, $A_{ij} = B_{ij} + C_{ij}$, where B and C's formats are $(\text{dense}_{d1}, \text{sparse}_{d2})$ and A's format is $(\text{dense}_{d1}, \text{dense}_{d2})$. (c–e) Internal representations used to generate the code. The algorithm and generated code have matching labels. The generated code is simplified in four ways: the outer loop is a for loop, if statements are nested in else branches, and if and min statements are removed from the last two while loops.

The last free variable is special because its loop nest is where the code writes to the output tensor. Loops nested above it compute available expressions (`emit-available-expressions`), which help avoid redundant computations. Loops nested below it are reduction loops that add sub-computations to reduction variables (`emit-reduction-compute`). Finally, the last free variable's loop combines the temporaries prepared by other loops to compute the final expression (`emit-compute`).

6.3 Index Assembly Code

The `emit-index-assembly` function emits code that assembles the index structure for sparse levels in the output tensor, which consists of a `pos` array that stores the start of each index segment and an `idx` array that contains indices for entries in the segments. For each of the output's sparse levels, code is emitted that inserts non-zero coordinates corresponding to that level into `idx`. For example,

```
A2_idx[pA2++] = j;
```

For levels above a sparse level in the output, `emit-index-assembly` also emits code to update the next level's `pos` array in order to match insertions into `idx`. However, a complication can arise above the last free variable as the sub-loops may not necessarily produce any value. In component-wise matrix multiplication, for instance, two rows might both have non-zeros but the intersection might not. To prevent empty locations in the result index structure, which is legal but sub-optimal in terms of compression, the emitted code checks if the sub-loops produced non-zeros. For example,

```
A2_pos[pA1+1] = pA2;
if (A2_pos[pA1+1] > A2_pos[pA1]) {
  A1_idx[pA1++] = i;
}
```

The conditional tests whether the current and previous `pos` for the sub-loops are the same. If not, the sub-loops produced non-zeros, so the assembly code inserts a new coordinate into the index.

Finally, it is necessary to allocate memory for the result tensor. This is handled by emitting code to check whether there is more space left in the `idx`, pos, and `vals` arrays before they are written to. If there is no more space then the emitted code doubles the memory allocation. Future work includes designing heuristics to set the initial size of the arrays.

6.4 Parallel Code Generation

The code generation algorithm annotates loops with OpenMP parallel pragmas. It only parallelizes outer loops, which provides sufficient parallelism and good parallel grain size. However, such heuristics should be controlled by the user. The loop parallelization is subject to three restrictions:

- The loop must not merge tensor dimensions, that is, it must be a `for` loop. This condition can be determined from merge lattices.
- Free variables cannot be dominated by reduction variables in the iteration graph, as this causes scatter behavior and we do not yet emit parallel synchronization constructs.
- The output tensor must be dense in all dimensions.

Future work includes relaxing these restrictions. Nevertheless, without relaxing them the code generation algorithm is still able to emit parallel code for many practical real-world kernels.

7 TACO C++ LIBRARY AND COMMAND-LINE TOOL

We have implemented the technique described in this paper in a compiler called `taco` (short for Tensor Algebra COmpiler). `taco` can be used as a C++ library that lets programmers compute on tensors within their applications. It can also be used as a command-line tool that lets users generate C kernels to include in their applications or to serve as the starting point for further development.[2]

Fig. 12 demonstrates how to use the C++ library to compute the tensor-times-vector multiplication from Section 2. `Tensor` objects can be created by specifying the dimensions of the tensor, the type of its entries, and the storage format. The storage format of a tensor can in turn be declared by creating a `Format` object describing the storage kind of each tensor level and the order in which levels are stored, following the formulation in Section 3. On lines 1–8, `A` is declared as a CSR

[2]The command-line tool can be accessed through a web interface at http://tensor-compiler.org/codegen.

```
1  Format csr({Dense,Sparse});
2  Tensor<double> A({64,42}, csr);
3
4  Format csf({Sparse,Sparse,Sparse});
5  Tensor<double> B({64,42,512}, csf);
6
7  Format svec({Sparse});
8  Tensor<double> c({512}, svec);
9
10 B.insert({0,0,0}, 1.0);
11 B.insert({1,2,0}, 2.0);
12 B.insert({1,2,1}, 3.0);
13 B.pack();
14
15 c.insert({0}, 4.0);
16 c.insert({1}, 5.0);
17 c.pack();
18
19 IndexVar i, j, k;
20 A(i,j) = B(i,j,k) * c(k);
21
22 A.compile();
23 A.assemble();
24 A.compute();
```

Fig. 12. Computing tensor-times-vector with the taco C++ library.

```
$taco "A(i,j) = B(i,j,k) * c(k)" -f=A:ds -f=B:sss -f=c:s
// ...
int pA2 = A2_pos[0];
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int i = B1_idx[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int j = B2_idx[pB2];
    double tk = 0.0;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while ((pB3 < B3_pos[pB2+1]) && (pc1 < c1_pos[1])) {
      int kB = B3_idx[pB3];
      int kc = c1_idx[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        tk += B_vals[pB3] * c_vals[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
    A_vals[pA2] = tk;
    pA2++;
  }
}
```

Fig. 13. Using the taco command-line tool to generate C code that computes tensor-times-vector. The output of the command-line tool is shown after the first line. Code to initialize tensors is elided.

matrix, B is declared as a CSF tensor, and c is declared as a sparse vector. Tensors that are inputs to computations can be initialized with user-defined data by specifying the coordinates and values of all non-zero components and then invoking the pack method to store the tensor in the desired storage format, as demonstrated on lines 10–17.

Tensor algebra computations are expressed in taco with tensor index notation, as shown on lines 19–20. Note the close resemblance between line 20 and the mathematical definition of tensor-times-vector multiplication presented in Section 2; this is achieved with operator overloading. Var objects correspond to index variables in tensor index notation. Summation reductions are implied over index variables that only appear on the right-hand side of an expression.

Calling compile on the target of a tensor algebra computation (A in the example) prompts taco to generate kernels to assemble index structures and compute. In the current implementation taco generates C code, calls the system compiler to compile it to a dynamic library, and then dynamically links it in with dlopen. This makes the functions available for later calls to assemble and compute. The system compiler handles low-level performance optimizations. These steps happen automatically and are not visible to users. This approach works well for development, but we plan to also implement an LLVM JIT backend that does not need a system compiler.

Next, the assemble method assembles the sparse index structure of the output tensor and preallocates memory for it. Finally, the actual computation is performed by invoking the compute method to execute the code generated by compile. By separating the output assembly from the actual computation, we enable users to assemble output tensors once and then recompute their values multiple times. This can be useful since, in many real-world applications, repeating a computation changes the values of the output tensor but not its non-zero structure.

Fig. 13 shows how to use the taco command-line tool to generate C code that computes the same tensor-times-vector operation. As before, tensor index notation is used to specify the operation that the generated kernel computes. The -f switch can be used to specify the storage format of the inputs and output, again following the formulation in Section 3. Under the hood, the command-line tool uses the same code generation mechanism as the C++ library. We used the taco command-line tool to generate all kernels described in this paper and include them as supplemental material.

Table 1. Summary of real-world matrices and higher-order tensors used in experiments.

Tensor	Domain	Dimensions	Non-zeros	Density
bcsstk17	Structural	10,974 × 10,974	428,650	4×10^{-3}
pdb1HYS	Protein data base	36,417 × 36,417	4,344,765	3×10^{-3}
rma10	3D CFD	46,385 × 46,385	2,329,092	1×10^{-3}
cant	FEM/Cantilever	62,451 × 62,451	4,007,383	1×10^{-3}
consph	FEM/Spheres	83,334 × 83,334	6,010,480	9×10^{-4}
cop20k	FEM/Accelerator	121,192 × 121,192	2,624,331	2×10^{-4}
shipsec1	FEM	140,874 × 140,874	3,568,176	2×10^{-4}
scircuit	Circuit	170,998 × 170,998	958,936	3×10^{-5}
mac-econ	Economics	206,500 × 206,500	1,273,389	9×10^{-5}
pwtk	Wind tunnel	217,918 × 217,918	11,524,432	2×10^{-4}
webbase-1M	Web connectivity	1,000,005 × 1,000,005	3,105,536	3×10^{-6}
Facebook	Social media	1591 × 63,891 × 63,890	737,934	1×10^{-7}
NELL-2	Machine learning	12,092 × 9184 × 28,818	76,879,419	2×10^{-5}
NELL-1	Machine learning	2,902,330 × 2,143,368 × 25,495,389	143,599,552	9×10^{-13}

8 EVALUATION

To evaluate the technique described in this paper, we use `taco` as well as several existing popular sparse linear and tensor algebra libraries to compute many practical expressions with real-world matrices and tensors as inputs. We demonstrate in Section 8.2 and Section 8.3 that our technique generates a wide range of sparse linear algebra kernels that are competitive with the performance of hand-optimized kernels, while eliminating the trade-off between performance and completeness that existing libraries make. We further demonstrate in Section 8.4 that these observations also hold true for sparse tensor algebra involving higher-order tensors. Finally, we show in Section 8.5 that support for a wide range of dense and sparse tensor storage formats is indeed essential for achieving high performance with real-world linear and tensor algebra computations.

8.1 Methodology

We evaluate `taco` against six existing widely used sparse linear algebra libraries. Eigen [Guennebaud et al. 2010], uBLAS [Walter and Koch 2007] and Gmm++ [Renard 2017] are C++ libraries that exploit template metaprogramming to specialize linear algebra operations for fast execution when possible. Eigen in particular has proven popular due to its high performance and relative ease of use, and it is used in many large-scale projects such as Google's TensorFlow [Abadi et al. 2016]. OSKI [Vuduc et al. 2005] is a C library that automatically tunes sparse linear algebra kernels to take advantage of optimizations such as register blocking and vectorization. pOSKI [Byun et al. 2012] is another library that is built on top of OSKI and implements a number of parallel optimizations from Williams [2007]. Intel MKL is a math processing library for C and Fortran that is heavily optimized for Intel processors [Intel 2012].

We also evaluate `taco` against two existing sparse tensor algebra libraries: SPLATT [Smith et al. 2015], a high-performance C++ toolkit designed primarily for sparse tensor factorization; and the MATLAB Tensor Toolbox [Bader and Kolda 2007], a MATLAB library that implements a number of sparse tensor factorization algorithms in addition to supporting primitive operations on general (unfactorized) sparse and dense higher-order tensors.

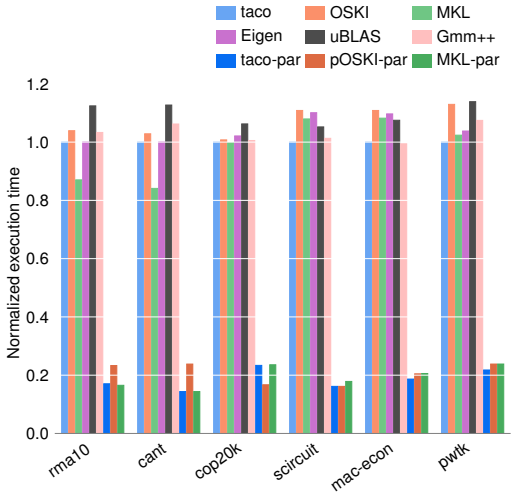


Fig. 14. Normalized execution time of SpMV with taco and existing sparse linear algebra libraries, relative to taco (running serially) for each matrix. -par denotes parallel results.

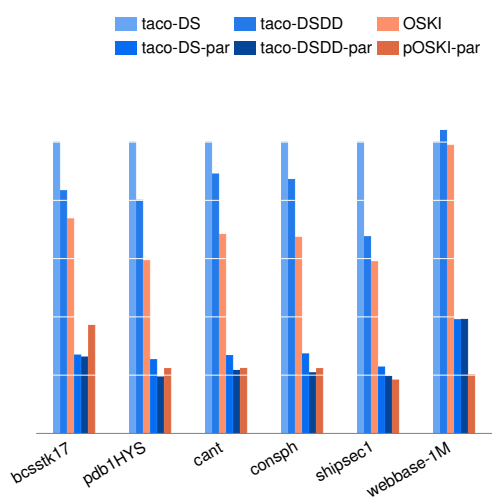


Fig. 15. Normalized execution time of blocked SpMV with taco, OSKI, and pOSKI (with tuning), relative to serial non-blocked SpMV with taco. -DSDD denotes blocked results.

The experiments described in Section 8.2 to Section 8.4 are run on real-world matrices and higher-order tensors obtained from the SuiteSparse Matrix Collection [Davis and Hu 2011] and the FROSTT Tensor Collection [Smith et al. 2017]. We also assemble a sparse tensor from a dataset of wall posts from the Facebook New Orleans regional network [Viswanath et al. 2009]. Table 1 reports some relevant statistics pertaining to these tensors.

We run all our experiments on a two-socket, 12-core/24-thread 2.4 GHz Intel Xeon E5-2695 v2 machine with 32 KB of L1 data cache, 30 MB of L3 cache per socket, and 128 GB of main memory, running MATLAB 2016b and GCC 5.4. We report average cold cache performance (i.e. with the cache cleared of input and output data before each run) and results are for single-threaded execution unless otherwise stated. Multi-threaded results were obtained using 12 threads and using `numactl` to limit execution to one socket as `taco` currently does not support NUMA-aware code generation.

8.2 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication is one of the most important operations in sparse linear algebra given its use in many applications. We measure `taco`-generated SpMV performance for matrices stored in the (dense_{d1},sparse_{d2}) format and compare against hand-coded SpMV kernels in existing sparse linear algebra libraries. For `taco`, SpMV is simply multiplying a 2nd-order tensor by a 1st-order tensor; we use the same code generation methodology for all tensor algebra kernels and do nothing to specifically optimize SpMV.

The results shown in Fig. 14 demonstrate that `taco` generates SpMV code with performance comparable to that of existing libraries when running on non-blocked matrices. On average, `taco` is competitive with Intel MKL and matches, if not slightly exceeds, single-threaded performance of all of the other libraries we evaluate. This is not very surprising as `taco` emits code that essentially implements the same high-level algorithm used by other libraries (i.e. stream through the non-zeros of the sparse matrix, scaling each by the corresponding element in the vector and accumulating

it into the output). In fact, the core of OSKI's SpMV kernel has virtually the same code structure as taco-emitted code. In the parallel case, taco is also able to match Intel MKL and pOSKI's performance on average despite implementing relatively simple parallelism. On this memory bandwidth-bound computation, taco obtains 72% of peak main memory bandwidth on average, compared to 70% for Intel MKL and 88% for pOSKI.

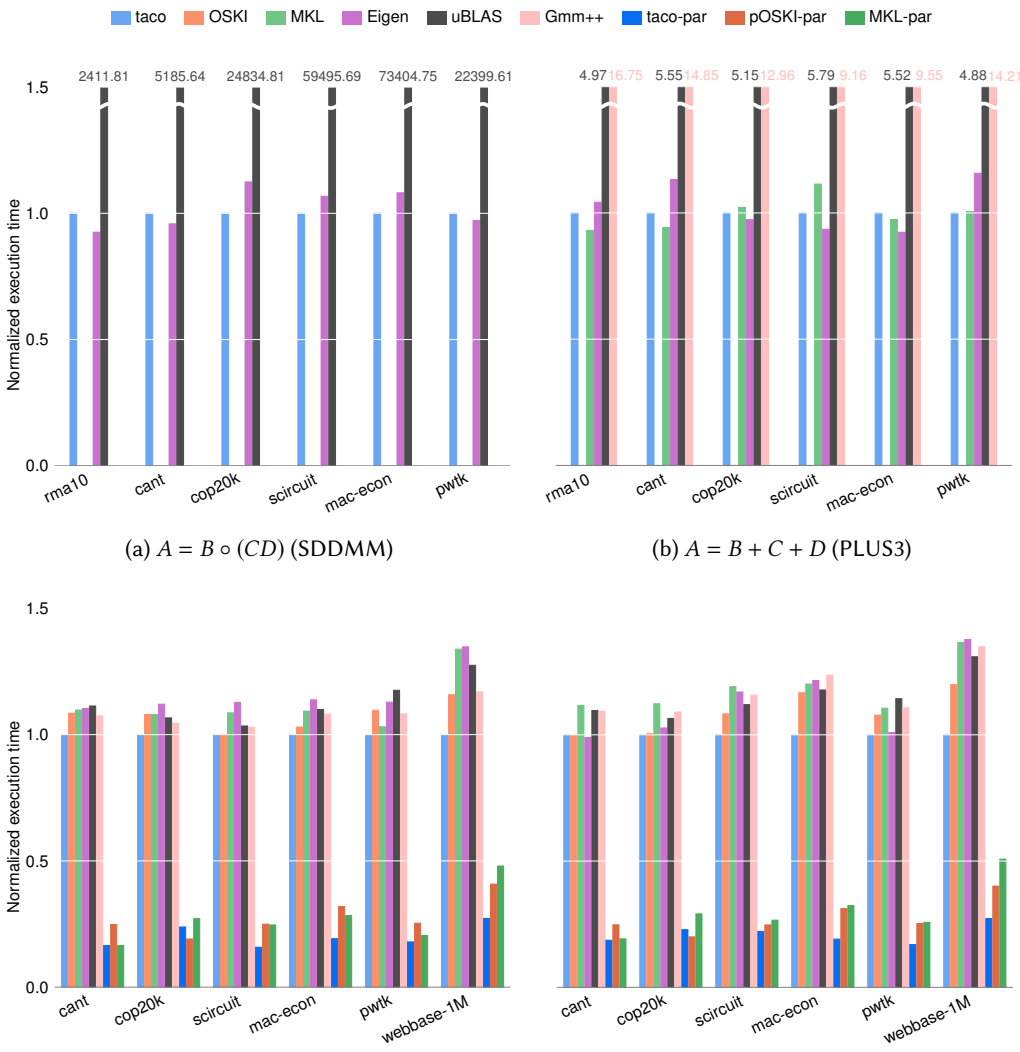
Matrices that originate from physical domains often contain small dense *blocks* of non-zeros. OSKI and pOSKI implement additional optimizations for such matrices using the block compressed sparse row (BCSR) format. In taco, the equivalent to this is a 4th-order tensor where the inner tensor dimensions use dense storage. Fig. 15 compares taco performance with tuned performance for OSKI and pOSKI. Tuned OSKI uses autotuning guided by a cost model to determine the best block size for each specific matrix; we obtain the block size used and run the equivalent taco-generated code. For the matrices that exhibit blocked structures, both OSKI and taco are able to significantly improve on the performance of regular SpMV by using the BCSR format. OSKI outperforms taco in almost all cases due to its carefully optimized and vectorized implementation. Nevertheless, in the parallel case, taco is again able to match pOSKI's performance on average. Overall, these results show that taco generates code competitive with hand-tuned libraries, even without yet implementing sophisticated optimizations for parallelism and vectorization.

8.3 Compound Linear Algebra

To demonstrate the impact of taco's ability to generate a single loop nest for compound linear algebra expressions, we compare its performance against existing libraries on four compound operations from real-world applications: SDDMM, PLUS3, MATTRANSMUL, and RESIDUAL. Fig. 16 shows the definitions of each operation as well as the results of the experiment. Only two of the sparse linear algebra libraries we compare taco against can compute all of four operations. In particular, apart from taco, only Eigen and uBLAS can compute SDDMM since the other libraries do not support component-wise multiplication of sparse matrices. In addition, OSKI and pOSKI do not support sparse matrix addition. In contrast, taco supports the full tensor index notation, including these compound operations, which demonstrates the versatility of our compiler technique.

The results in Fig. 16 also show that taco is comparable to, if not better than, existing sparse linear algebra libraries in terms of serial and parallel performance for compound operations. For some benchmarks, we observe significant performance improvements relative to existing libraries. For instance, Gmm++ is about an order of magnitude slower than taco for PLUS3 while uBLAS is up to several orders of magnitude slower than taco for SDDMM. taco also matches or exceeds the performance of all of the libraries we evaluate for MATTRANSMUL and RESIDUAL, including OSKI and Intel MKL, both of which implement hand-optimized kernels that directly compute the sum of a vector and the result of a sparse matrix-vector product. This result holds true on average in the parallel case as well, demonstrating that taco is capable of generating parallel kernels that provide additional performance on modern multicore processors for many expressions. taco-generated C kernels that compute these compound expressions contain between 30 lines of code (for RESIDUAL) to as many as 240 lines of code (for PLUS3), which suggests that it can be tedious and labor-intensive to implement similar kernels by hand, especially if a library developer also has to implement many variations of the same kernel for different tensor storage formats and architectures.

In the case of SDDMM, taco is able to minimize the number of floating-point operations needed for the computation by taking advantage of the last optimization listed in Section 5.2 to avoid computing components of the intermediate dense matrix product that cannot possibly contribute non-zeros to the output (i.e. the components with coordinates that correspond to zeros in the sparse matrix operand). uBLAS, in contrast, effectively computes the entire dense matrix product and thereby does $\Theta(n^3)$ work as opposed to $\Theta(n \cdot \text{nnz}(B))$ for n-by-n inputs.



(a) $A = B \circ (CD)$ (SDDMM)

(b) $A = B + C + D$ (PLUS3)

(c) $y = \alpha A^\top x + \beta z$ (MATTRANSMUL)

(d) $y = b - Ax$ (RESIDUAL)

Fig. 16. Normalized execution time of four compound sparse linear algebra operations, relative to taco for each operation and matrix. Missing bars denote that the operation is not supported by that library. We omit parallel results for SDDMM and PLUS3 since none of the libraries support parallel execution for those operations; taco support for parallel operations with sparse results is work in progress. Time to assemble output indices is included in PLUS3 results. Other than the SDDMM matrices C and D (which are dense), all matrices are sparse in the CSR or CSC format. All vectors are dense.

Our approach also generates fused compound kernels that avoid large intermediate results. For example, to compute MATTRANSMUL or RESIDUAL with uBLAS, one must compute and store the sparse matrix-vector product in a temporary vector and then compute the final result. Similarly, with Intel MKL or OSKI, one must first copy the input vector z to the output vector, in essence treating the output as a temporary vector, and then call the SpMV kernel. The reason is that these

Table 2. Normalized execution time of various sparse tensor algebra kernels, relative to taco for each kernel and tensor. A missing entry means the kernel is not supported by that library while OOM denotes that the kernel does not execute due to lack of memory. We report serial results for all kernels in addition to parallel results (denoted by -par) for MTTKRP. (As the MATLAB Tensor Toolbox does not support parallel MTTKRP, we compare their serial implementation against taco's parallel implementation.) All 3rd-order tensors use the (sparse$_{d1}$,sparse$_{d2}$,sparse$_{d3}$) format, while all input matrices and vectors are dense.

	Facebook			NELL-2			NELL-1		
	taco	TTB	SPLATT	taco	TTB	SPLATT	taco	TTB	SPLATT
TTV	1	65.74		1	103.9		1	15.49	
TTM	1	255.0		1	9.329		1	OOM	
MTTKRP	1	14.06	1.218	1	29.17	0.7157	1	11.24	0.8371
MTTKRP-par	1	84.15	1.266	1	318.8	0.7826	1	76.48	0.6542
PLUS	1	39.47		1	266.7		1	179.6	
INNERPROD	1	113.6		1	1856		1	1509	

libraries only support incrementing the output (and not some arbitrary input vector) by the result of a matrix-vector product. Thus, the corresponding temporary vector has to be scanned twice, which results in increased memory traffic if the vector is too large to fit in cache. taco, on the other hand, generates code for MATTRANSMUL and RESIDUAL that compute each component of the matrix-vector product as needed when evaluating the top-level vector summation. This avoids the temporary vector and reduces memory traffic with larger inputs such as webbase-1M. This further shows the advantage of a compiler approach that generates kernels for the computation at hand.

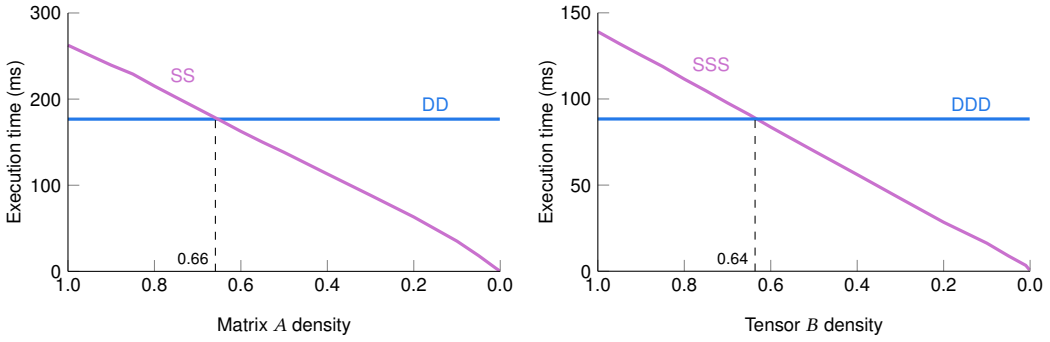
8.4 Higher-Order Tensor Algebra

To demonstrate that our technique is not restricted to the linear algebra subset of tensor algebra, we compare performance for the following set of sparse 3rd-order tensor algebra kernels:

TTV $A_{ij} = \sum_k B_{ijk} c_k$
TTM $A_{ijk} = \sum_l B_{ijl} C_{kl}$
MTTKRP $A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$
PLUS $A_{ijk} = B_{ijk} + C_{ijk}$
INNERPROD $\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

All of these operations have real-world applications. TTM and MTTKRP, for example, are important building blocks of algorithms to compute the Tucker and canonical polyadic (CP) decompositions [Smith et al. 2015]. Table 2 shows the results of this experiment. SPLATT and the MATLAB Tensor Toolbox (TTB) exemplify different points in the tradeoff space for hand-written libraries: SPLATT only supports one of the kernels we evaluate but executes it efficiently, while the Tensor Toolbox supports all of the evaluated tensor operations but does not achieve high performance. (In this section, we do not consider any of the sparse linear algebra libraries we looked at in the previous two sections as none of them supports sparse higher-order tensor algebra.) Meanwhile, taco supports all five operations and generates efficient code for each, illustrating our technique's versatility.

Code generated by taco outperforms equivalent Tensor Toolbox kernels by at least about an order of magnitude on all of our benchmarks. These significant performance differences are the direct consequence of the Tensor Toolbox's general approach to sparse tensor algebra computation, which heavily relies on functionality built into MATLAB for performance. In order to compute TMM for instance, the Tensor Toolbox first matricizes the sparse tensor input, storing the result as a



(a) $y = \alpha Ax + \beta z$ (b) $A_{ij} = \sum_k B_{ijk} c_k$

Fig. 17. Average execution times of (a) matrix-vector multiplication and (b) tensor-times-vector multiplication, with 15,000 × 15,000 matrices and 500 × 500 × 500 tensors of varying densities in sparse and dense formats.

CSC matrix, and then takes advantage of MATLAB's native support for sparse matrices to perform the actual computation as a sparse matrix-dense matrix multiplication. However, matricizing the sparse tensor input as well as converting the output of the matrix multiplication (which is stored as a dense matrix Z) back to a sparse tensor are both very costly operations that can actually take as much time to execute as the matrix multiplication itself. Additionally, since one dimension of Z corresponds to the Cartesian product of all unreduced dimensions in the input tensor, the kernel can run out of memory if the dimensions of the input are too large, as we observed with NELL-1 for example. In contrast, since `taco` emits code that directly computes on the sparse tensor input and assembles the sparse output directly without needing the dense temporary storage, `taco` is able to avoid the format conversion overhead.

The serial and parallel performance of `taco`-emitted MTTKRP is also competitive with that of SPLATT's hand-optimized implementation, which shares a similar high-level loop structure but reduces the number of floating-point operations needed to compute MTTKRP for larger inputs by essentially reformulating the computation as $A_{ij} = \sum_k \sum_l B_{ikl} C_{kj} D_{lj} = \sum_k C_{kj} (\sum_l B_{ikl} D_{lj})$, factoring out the repeated multiplications. While `taco` currently does not perform this optimization, it nevertheless outperforms SPLATT on the Facebook tensor benchmarks and is never slower than SPLATT by more than 1.53× on any of the remaining benchmarks, demonstrating that performance and flexibility need not be mutually exclusive in sparse tensor algebra libraries.

We also evaluate the performance of MTTKRP with sparse input matrices against that of the same computation with dense matrices as input. To measure this, we run MTTKRP with randomly generated $I \times 25$ matrices (where I denotes the size of the first dimension of tensor B) containing 25 non-zeros per column. `taco` is able to take advantage of the sparsity of the input matrices by storing them as (sparse_{d2},sparse_{d1}) matrices and generating a kernel that directly computes on the sparse data structures. This enables `taco` to obtain speedups ranging from 1.66× (for NELL-1) to 55.7× (for NELL-2) relative to dense MTTKRP, highlighting the benefit of a compiler approach that shapes tensor algebra code to data stored in efficient formats.

8.5 Choice of Formats

While most of the real-world tensors we looked at are highly sparse, in many applications using sparse tensor storage formats to store much denser data can still offer tangible performance benefits. To illustrate this, we compare the performance of dense and sparse matrix-vector multiplication

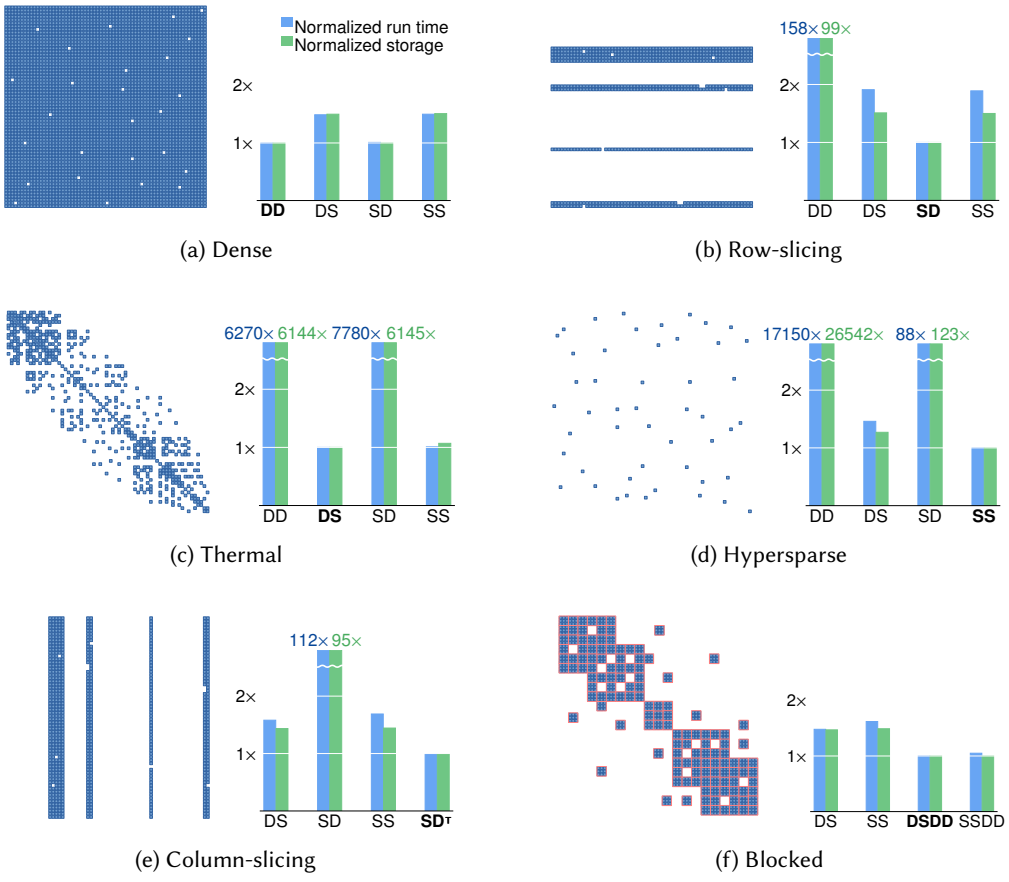


Fig. 18. Performance of matrix-vector multiplication on various matrices with distinct sparsity patterns using taco. The left half of each subfigure depicts the sparsity pattern of the matrix, while the right half shows the normalized storage costs and normalized average execution times (relative to the optimal format) of matrix-vector multiplication using the storage formats labeled on the horizontal axis to store the matrix. The storage format labels follow the scheme described in Section 3; for instance, DS is short for (dense$_{d1}$,sparse$_{d2}$), while SD$^\intercal$ is equivalent to (sparse$_{d2}$,dense$_{d1}$). The dense matrix input has a density of 0.95, the hypersparse matrix has a density of 2.5×10^{-5}, the row-slicing and column-slicing matrices have densities of 9.5×10^{-3}, and the thermal and blocked matrices have densities of 1.0×10^{-3}.

and tensor-times-vector multiplication with matrices and 3rd-order tensors of varying sparsities as inputs. The tensors are randomly generated with every component having some probability d of being non-zero, where d is the density of the tensor (i.e. the fraction of components that are non-zero, and the complement of sparsity). As Fig. 17 shows, while computing with sparse tensor storage formats incurs some performance penalty as compared to the same computation with dense formats when the inputs are highly dense, the performance penalty decreases and eventually turns into performance gain as the sparsity of the inputs increases. For the two computations we evaluate, we observe that input sparsity of as low as approximately 35% is actually sufficient to make sparse formats that compress out all zeros—including DCSR and CSF—perform better than dense formats, which further emphasizes the practicality of sparse tensor storage formats.

Traditional sparse tensor and linear algebra libraries are forced to limit the set of storage formats they support because of the difficulty of hand-coding each operation for each format combination. For instance, Eigen does not support blocked formats, while pOSKI supports only CSR and BCSR matrices; for higher-order tensors, SPLATT only supports the CSF format for most operations. However, even for a particular tensor algebra expression, the storage format that gives the best performance depends on the sparsity and the structure of the input tensor. To demonstrate this, we consider six classes of real-world matrices with very different structures; the typical sparsity patterns of these matrices are shown on the left of each subfigure of Fig. 18. For each class of matrices, we generate a random matrix with that sparsity pattern and measure memory usage and matrix-vector multiplication performance for various storage formats with `taco`.

The results of this experiment, which are shown on the right of each subfigure of Fig. 18, make it clear there is no ideal tensor storage format that outperforms all alternatives in general. For each matrix type, we find there is a distinct format that minimizes storage cost and yields the highest performance for matrix-vector multiplication. Additionally, choosing an improper storage format can significantly increase both the storage and computation costs. Storing the thermal matrix using the (sparse$_{d1}$,dense$_{d2}$) format, for instance, increases memory usage and matrix-vector multiplication execution time relative to optimal by several orders of magnitude. This is because the (sparse$_{d1}$,dense$_{d2}$) format ends up essentially having to store all components of the thermal matrix, including all of the zeros, which in turn increases the asymptotic complexity of the matrix-vector multiplication from $\Theta(\text{nnz})$ to $\Theta(n^2)$. For the same reason, the (sparse$_{d1}$,dense$_{d2}$) format turns out to be a poor fit for column-slicing matrices despite being the optimal format for row-slicing matrices with the same sparsity, highlighting the advantages of being able to store dimensions of a tensor in arbitrary orders. `taco`, which generates code that are tailored to specific tensor storage formats, makes it simple to compute using the formats that best suit the input data.

9 RELATED WORK

Prior work on linear algebra goes back to the earliest days of computing. Recently, researchers have also explored tensor algebra. In this section, we group linear and tensor algebra together and discuss their dense and sparse variants individually. We discuss the work on sparse linear algebra compilers separately since it is closest to our work.

9.1 Dense Linear and Tensor Algebra Libraries, Languages and Compilers

There has been a lot of work on languages [Bezanson et al. 2012; Iverson 1962; MATLAB 2014], libraries [Anderson et al. 1999; Guennebaud et al. 2010; Intel 2012; Sanderson 2010; Van Der Walt et al. 2011; Whaley and Dongarra 1998], and compilers [Nelson et al. 2015; Spampinato and Püschel 2014] for dense linear algebra and loop transformations that can optimize dense loop nests [McKinley et al. 1996; Wolf and Lam 1991; Wolfe 1982].

In the last decade researchers have also explored dense tensor algebra. Most efforts have focused on tensor contractions in quantum chemistry, but recently tensor operations in data analytics and machine learning have received attention. These works share many similarities, but the domains require support for different tensor properties, operations and execution strategies. In quantum chemistry, an early effort was the Tensor Contraction Engine [Auer et al. 2006], which is a compiler framework developed to automatically optimize dense tensor contractions (multiplications) in the NWChem software. Mullin and Reynolds [2014] proposed an algebraic methodology for augmenting MATLAB to efficiently execute Kronecker products and other tensor operations by generating code that uses dense array operations based on a Mathematics of Arrays and the ψ-calculus [Mullin 1988]. Two recent efforts are libtensor with extensive support for tensor symmetries [Epifanovsky et al. 2013]; and CTF with a focus on distributed computations [Solomonik et al. 2014]. Both

libtensor and CTF turn tensor contractions into matrix multiplications by transposing the tensors. The GETT library optimizes this process by leveraging a fast transposition engine [Springer and Bientinesi 2016], and the BLIS framework by fusing transpositions with later stages [Matthews 2017]. In contrast, the InTensLi framework avoids transpositions by computing tensor-times-matrix operations in-place [Li et al. 2015]. In machine learning, TensorFlow is a recent framework where dense tensors are passed between tensor computation kernels in a dataflow computation [Abadi et al. 2016]. Finally, Cai et al. [2015] explores how to optimize the MTTKRP operation for symmetric tensors with applications to data analytics.

9.2 Sparse Linear and Tensor Algebra Libraries and Languages

The use of sparse matrix data structures goes back to Tinney and Walker [1967] and a library described by McNamee [1971]. Gustafson [1978] first described the sparse matrix-matrix multiplication algorithm in use today. More recently MATLAB [2014], Julia [Bezanson et al. 2012], Eigen [Guennebaud et al. 2010], and PETSc [Balay et al. 1997] have become popular for computing with sparse matrices. MATLAB, Eigen, and Julia are general systems that support all basic linear algebra operations; however, their sparse matrix formats are limited. PETSc targets supercomputers and supports distributed and blocked matrix formats. OSKI and the parallel pOSKI are well-known libraries that support auto-tuning of some sparse kernels [Byun et al. 2012; Vuduc et al. 2005]. However, the feature set is limited to SpMV, triangular solve, matrix powers, and simultaneously multiplying a matrix and its transpose by vectors.

Furthermore, Buluç et al. [2009] described a scalable parallel implementation of SpMV where the matrix can be transposed or not. The implementation uses a new format called compressed sparse blocks (CSB) that consists of a dense matrix with sparse blocks stored as coordinates. Our approach does not yet support coordinates, but it can express an analogous format with two dense outer dimensions and two sparse inner dimensions. In addition, by further nesting two dense dimensions inside the sparse dimensions, it can express the combined CSB/BCSR format they conjecture.

The MATLAB Tensor Toolbox is an early system for sparse tensors that supports many tensor operations with the coordinate format and other formats for factorized tensors [Bader and Kolda 2007]. Recently, Solomonik and Hoefler [2015] described a sparse version of the distributed CTF library. These systems convert sparse tensors to sparse matrices and then call hand-coded sparse matrix routines. Other researchers have developed dedicated sparse tensor kernels that avoid this translation overhead. These include SPLATT, which supports fast shared-memory parallel MTTKRP and tensor contractions [Smith et al. 2015], HyperTensor, which supports distributed MTTKRP [Kaya and Uçar 2015], an exploration into reuse optimizations for a mixed sparse-dense format by Baskaran et al. [2012], and a fast shared-memory and GPU parallel tensor-times-dense matrix multiply by Li et al. [2016]. Finally, TensorFlow recently added support for some sparse tensor operations as hand-coded kernels on tensors stored in the coordinate format [Google 2017].

In contrast to these approaches, we describe how to automatically generate sparse tensor algebra kernels instead of hand-coding them. The benefit is that we efficiently support any compound expression with many different storage formats.

9.3 Sparse Linear and Tensor Algebra Compilers

Prior work on sparse linear algebra compilers is most related to our approach. Several researchers have proposed techniques to compile dense linear algebra loops to sparse linear algebra loops. Bik and Wijshoff [1993; 1994] developed an early compiler that turns dense loops over dense arrays into sparse loops over the non-zero array values, using a technique they call guard encapsulation to move non-zero guards into sparse data structures. Thibault et al. [1994] described compiler

techniques to generate indexing functions for sparse arrays when the sparsity is regular, enabling compressed storage and efficient indexing.

The Bernoulli project [Kotlyar et al. 1997] reduced declarative constraint expressions that enumerate sparse iteration spaces to relational algebra queries by converting sparsity guards into predicates on relational selection and join expressions. This avoids the need to find the sequence of loop transformations that puts the code in the correct form for guard encapsulation. They then build on techniques from the database literature to optimize queries and insert efficient join implementations. Their work focuses on sparse linear algebra and supports conjunctive loops without merges, such as SpMV [Kotlyar 1999, Introduction]. They conjecture that their approach can be extended to disjunctive binary merges (e.g., matrix addition) by casting them as outer joins [Stodghill 1997, Chapter 15.1], but did not explore this further [Kotlyar 1999, Chapter 11.2].

SIPR [Pugh and Shpeisman 1999] is an intermediate representation for sparse matrix operations that transforms dense code to sparse code. SIPR supports row swaps, but does not address index merging in a general way, restricting the possible operations. LL [Arnold 2011] is a small language designed for functional verification of sparse formats and can generate code for binary sparse operations as well as verify their correctness. However, LL does not generate code for compound linear algebra. Venkat et al. presented the transformations *compact* and *compact-and-pad* which turn dense loops with conditional guards into loops over a sparse matrix in one of several formats [Venkat et al. 2015]. However, they do not discuss loops with more than one sparse matrix, which require merging indices. Further improvements to the compiler and runtime framework enable automatically applying wavefront parallelism in the presence of loop-carried dependencies [Venkat et al. 2016]. Recently, Sparso demonstrated that context can be exploited to optimize sparse linear algebra programs by reordering matrices and taking advantage of matrix properties [Rong et al. 2016]. These optimizations are orthogonal to our technique and the two can reinforce each other.

By contrast, our approach generalizes to sparse tensor algebra and supports compound expressions. Furthermore, we start from index expressions instead of loops, which frees us from the need to perform complex control-flow analyses.

10 CONCLUSION AND FUTURE WORK

We have described the first technique for compiling compound tensor algebra expressions with dense and sparse operands to fast kernels. Three ideas came together in a simple code generation algorithm. First, we showed how tensor storage formats can be composed from simple building blocks that designate each dimension as dense or sparse, along with a specification of the order in which dimensions should be stored. Second, we introduced iteration graphs that capture the dependencies between nested loops due to the tensor storage formats. Third, we developed merge lattices that generalize the idea behind two-way merge loops to yield fast code that merges all the tensor indices accessed by a tensor index variable. The performance of the generated code shows that it is possible to get both generality and performance in the same system.

With these ideas, we can develop libraries and languages for dense and sparse tensor algebra without hand-writing kernels or sacrificing performance. To promote this, we have released `taco` under the MIT license at http://tensor-compiler.org. We also believe some concepts in this paper apply to other domains. For example, the merge lattices can be used in any domain where data structures need to be merged, such as merge sort and database joins (inner joins are conjunctive and outer joins are disjunctive).

The ideas in this paper put sparse tensor algebra on a firm theoretical and practical compiler foundation. We foresee many directions of future work, including:

Portability The current work produces shared-memory parallel code with restrictions described in Section 6.4. Future work includes removing these restrictions and adapting the code generation to target accelerators (e.g., GPUs and TPUs) and distributed memory systems (e.g., supercomputers and data centers). With such extensions, the ideas in this paper form the basis for a portable system that can generate kernels tuned to each machine.

Formats We present two formats per dimension, dense and sparse, but many more are possible. We are particularly interested in the coordinate format, as it is the natural input format with no packing cost. Other possibilities include diagonal formats, hashed formats, and other formats that support fast modification. To manage complexity, we envision a plugin system that makes it easy to add and combine formats.

Transposes The code generation algorithm in Section 6 does not support cycles in iteration graphs that result from accessing tensors in the opposite direction of their formats, and format conversions are sometimes necessary. Future work includes generalizing the code generation algorithm to support cycles, thus removing the need for conversion.

Workspaces Kernels with sparse outputs that may write several times to each location, such as linear combination of columns sparse matrix-matrix multiplication, benefit from a workspace tensor with random access. Developing a theory for breaking up iteration graphs and introducing workspaces is interesting future work.

Tiling The formats we present in this paper can express data blocking, such as the blocked sparse matrices in Section 8.2. Future work includes generalizing iteration graphs and code generation to also support iteration space blocking/tiling. This is necessary for the performance of dense kernels when the data cannot be blocked.

Semirings The concepts and techniques in this paper are compatible with any semiring. However, it remains to investigate how to handle cases with mixed operations, such as taking the product reduction of a vector while working in the normal $(+, *)$ semiring.

Autotuning We believe in separating policy (deciding what to do) from mechanism (doing it). Since policy depends on mechanism, we have focused on pure mechanism in this paper. However, given mechanism, future work can explore how to autotune choice of storage formats and expression granularity. This includes search, model-driven optimization, machine learning, and heuristics.

Runtime Support Many properties of sparse tensors, such as size, density, and shape (e.g., symmetry and block sizes), are only available at runtime. We envision a runtime system that takes advantage of them through intelligent runtime analysis and JIT compilation.

Formalism This paper presents new compiler concepts and algorithms. An avenue for future work is to formally define these and to prove correctness and completeness properties.

Taken together, we envision a portable system that shapes computation to data structures as they are, with no need for costly data translation. We believe that code should adapt to data, so that data may stay at rest.

ACKNOWLEDGMENTS

We thank Peter Ahrens, Jeffrey Bosboom, Gurtej Kanwar, Vladimir Kiriansky, Charith Mendis, Jessica Ray, Yunming Zhang, and the anonymous reviewers for helpful reviews and suggestions. This work was supported by the National Science Foundation under Grant No. CCF-1533753, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC008923 and DE-SC014204, and by the Toyota Research Institute. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. 2014. Tensor Decompositions for Learning Latent Variable Models. *J. Mach. Learn. Res.* 15, Article 1 (Jan. 2014), 60 pages.

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Gilad Arnold. 2011. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes*. Ph.D. Dissertation. University of California, Berkeley.

Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2006. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics* 104, 2 (2006), 211–228.

Brett W. Bader, Michael W. Berry, and Murray Browne. 2008. *Discussion Tracking in Enron Email Using PARAFAC*. Springer London, 147–163.

Brett W Bader and Tamara G Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2007), 205–231.

Satish Balay, William D Gropp, Lois Curfman McInnes, and Barry F Smith. 1997. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern software tools for scientific computing*. Springer, Birkhäuser Boston, 163–202.

Muthu Baskaran, Benoît Meister, Nicolas Vasilache, and Richard Lethin. 2012. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*. IEEE, 1–6.

James Bennett, Stan Lanning, et al. 2007. The netflix prize. In *Proceedings of KDD cup and workshop*, Vol. 2007. ACM, New York, 35.

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. (2012).

Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. ACM, 416–424.

Aart JC Bik and Harry AG Wijshoff. 1994. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing*. Springer, 57–75.

Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS)*. 1–11.

Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

Jong-Ho Byun, Richard Lin, Katherine A Yelick, and James Demmel. 2012. Autotuning sparse matrix-vector multiplication for multicore. *EECS, UC Berkeley, Tech. Rep* (2012).

Jonathon Cai, Muthu Baskaran, Benoît Meister, and Richard Lethin. 2015. Optimization of symmetric tensor computations. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 1–7.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011).

Albert. Einstein. 1916. The Foundation of the General Theory of Relativity. *Annalen der Physik* 354 (1916), 769–822.

Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I Krylov. 2013. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry* 34, 26 (2013), 2293–2309.

Richard Feynman, Robert B. Leighton, and Matthew L. Sands. 1963. *The Feynman Lectures on Physics*. Vol. 3. Addison-Wesley.

Google. 2017. TensorFlow Sparse Tensors. https://www.tensorflow.org/api_guides/python/sparse_ops. (2017).

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (1978).

Intel. 2012. *Intel math kernel library reference manual.* Technical Report. 630813-051US, 2012. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf.

Kenneth E. Iverson. 1962. *A Programming Language.* Wiley.

Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, 77.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graphics* (2016).

Donald Ervin Knuth. 1973. *The art of computer programming: sorting and searching.* Vol. 3. Pearson Education.

Joseph C Kolecki. 2002. An Introduction to Tensors for Students of Physics and Engineering. *Unixenguaedu* 7, September (2002), 29.

Vladimir Kotlyar. 1999. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs.* Ph.D. Dissertation. Cornell.

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing.* Springer, 318–327.

Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* ACM, 76.

Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms.* IEEE Press, 26–33.

MATLAB. 2014. *version 8.3.0 (R2014a).* The MathWorks Inc., Natick, Massachusetts.

Devin Matthews. 2017. *High-Performance Tensor Contraction without Transposition.* Technical Report.

Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems.* ACM, 165–172.

Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 4 (1996), 424–453.

John Michael McNamee. 1971. Algorithm 408: a sparse matrix package (part I)[F4]. *Commun. ACM* 14, 4 (1971), 265–273.

Lenore Mullin and James Raynolds. 2014. *Scalable, Portable, Verifiable Kronecker Products on Multi-scale Computers.* Springer International Publishing, Cham, 111–129.

L. M. R. Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation. Syracuse University.

Thomas Nelson, Geoffrey Belter, Jeremy G. Siek, Elizabeth Jessup, and Boyana Norris. 2015. Reliable Generation of High-Performance Matrix Algebra. *ACM Trans. Math. Softw.* 41, 3, Article 18 (June 2015), 27 pages.

William Pugh and Tatiana Shpeisman. 1999. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing.* Springer, 213–229.

Yves Renard. 2017. Gmm++. (2017). http://download.gna.org/getfem/html/homepage/gmm/first-step.html

Gregorio Ricci-Curbastro and Tullio Levi-Civita. 1901. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54 (1901).

Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation.* ACM, 247–259.

Conrad Sanderson. 2010. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical Report. NICTA.

Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. (2017). http://frostt.io/

Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms.* ACM, 5.

Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 61–70.

Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv preprint arXiv:1512.00066* (2015).

Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

Daniele G Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization.* ACM, 23.

Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *arXiv preprint arXiv:1607.00145* (2016).

Paul Stodghill. 1997. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes.* Ph.D. Dissertation. Cornell.

Scott Thibault, Lenore Mullin, and Matt Insall. 1994. Generating Indexing Functions of Regularly Sparse Arrays for Array Compilers. (1994).

William F Tinney and John W Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. 521–532.

Anand Venkat, Mahdi Soltan Mohammadi, Hongbo Rong, Rajkishore Barik, Jongsoo Park, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *In Supercomputing (SC)*.

Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the Evolution of User Interaction in Facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 1 (2005), 521+.

Joerg Walter and Mathias Koch. 2007. uBLAS. (2007). http://www.boost.org/libs/numeric/ublas/doc/index.htm

R. Clint Whaley and Jack Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. New York, NY, USA, 38:1–38:12.

Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. *SIGPLAN Not.* 26, 6 (May 1991), 30–44.

Michael Joseph Wolfe. 1982. *Optimizing Supercompilers for Supercomputers.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA. AAI8303027.

Huasha Zhao. 2014. *High Performance Machine Learning through Codesign and Rooflining.* Ph.D. Dissertation. EECS Department, University of California, Berkeley.