The Tensor Algebra Compiler

Fredrik Kjolstad
Massachusetts Institute of Technology
fred@csail.mit.edu

Shoaib Kamil
Adobe Research
kamil@adobe.com

Stephen Chou
Massachusetts Institute of Technology
s3chou@csail.mit.edu

David Lugato
French Alternative Energies and
Atomic Energy Commission
david.lugato@cea.fr

Saman Amarasinghe
Massachusetts Institute of Technology
saman@csail.mit.edu

Abstract

Tensor and linear algebra is pervasive in data analytics and the physical sciences. Often the tensors, matrices or even vectors are sparse. Computing expressions involving a mix of sparse and dense tensors, matrices and vectors requires writing kernels for every operation and combination of formats of interest. The number of possibilities is infinite, which makes it impossible to write library code for all. This problem cries out for a compiler approach. This paper presents a new technique that compiles compound tensor algebra expressions combined with descriptions of tensor formats into efficient loops. The technique is evaluated in a prototype compiler called `taco`, demonstrating competitive performance to best-in-class hand-written codes for tensor and matrix operations.

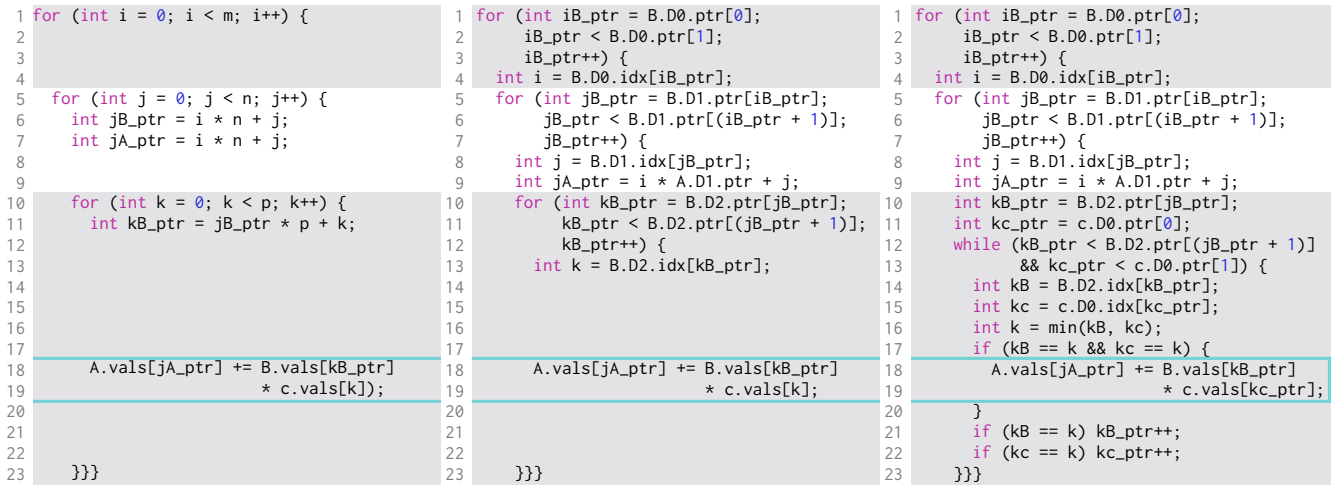
1. Introduction

Dense linear algebra is a powerful and ubiquitous tool, and many libraries, languages and compilers have been built to support it. However, many real-world problems are sparse and it is wasteful or intractable to store the zero values. Furthermore, many phenomena are multi-dimensional and benefit from the generalization of linear algebra to tensor algebra. Tensors generalize vectors and matrices to more dimensions and have applications in science [19, 42], engineering [18, 24] and data analytics [4, 10]. For example, many large real-world data sets used in big data analysis are large sparse tensors, such as Netflix ratings [12] and Facebook activities [45]. The number of algorithms for such data sets are growing and they require high performance, which means compiler support for optimizing tensor algebra expressions and operations on sparse data is of utmost importance.

A number of languages [13, 22, 28], libraries [2, 5, 20, 36, 43, 48], and compilers [31, 39] have been developed to support dense linear algebra. These libraries focus on providing fast implementations of the most highly-used operations. In contrast, compiled languages optimize complex compound linear algebra statements with multiple operations [29, 49, 50]. However, there have been only a few systems that support either sparse matrix operations or dense tensor operations and even fewer for sparse tensor computations. Most sparse matrix applications use libraries [11, 20, 32, 46], though some compilers [14, 26, 44] do support sparse linear algebra. More recently, new libraries [3] and compilers [7] for dense tensors are emerging. Libraries for sparse tensors are also being developed [9, 37]. However, to the best of our knowledge, no high performance compiler exists for sparse tensor algebra.

This paper presents the first technique that generates efficient code for any compound tensor and linear algebra expression, where the operands are dense, sparse or mixed. In order to represent different tensor formats, we develop a unified representation. In addition, we define an intermediate representation for compound tensor expressions. Using these two, we present a code generation algorithm. Finally, we implement these techniques in a compiler. The main contributions of this paper are:

- A *Tensor Storage* representation that recursively defines the storage of multi-dimensional tensors. Each dimension can be stored using a dense or sparse layout, which lets us map sparse data sets into highly-efficient compact memory layouts. This storage representation encompasses many widely-used matrix and tensor formats (§ 3).
- An *Iteration Schedule* intermediate representation describing how to iterate through the multi-level sparse iteration space of any compound tensor algebra expression (§ 4).
- A *Merge Lattice* representation that lets us generate efficient code for merge iteration spaces (§ 5.1).
- A *Code Generation Algorithm* that translates an iteration schedule to efficient code that evaluates the corresponding tensor algebra expression through a single pass over the sparse iteration space of its operands (§ 5.2–5.3).



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

(a) Dense A_{ij}, B_{ijk}, c_k

```
1  for (int iB_ptr = B.D0.ptr[0];
2       iB_ptr < B.D0.ptr[1];
3       iB_ptr++) {
4    int i = B.D0.idx[iB_ptr];
5    for (int jB_ptr = B.D1.ptr[iB_ptr];
6         jB_ptr < B.D1.ptr[(iB_ptr + 1)];
7         jB_ptr++) {
8      int j = B.D1.idx[jB_ptr];
9      int jA_ptr = i * A.D1.ptr + j;
10     for (int kB_ptr = B.D2.ptr[jB_ptr];
11          kB_ptr < B.D2.ptr[(jB_ptr + 1)];
12          kB_ptr++) {
13       int k = B.D2.idx[kB_ptr];
14
15
16
17
18       A.vals[jA_ptr] += B.vals[kB_ptr]
19                       * c.vals[k];
20
21
22
23   }}}
```

(b) Sparse B_{ijk}, Dense A_{ij} and c_k

```
1  for (int iB_ptr = B.D0.ptr[0];
2       iB_ptr < B.D0.ptr[1];
3       iB_ptr++) {
4    int i = B.D0.idx[iB_ptr];
5    for (int jB_ptr = B.D1.ptr[iB_ptr];
6         jB_ptr < B.D1.ptr[(iB_ptr + 1)];
7         jB_ptr++) {
8      int j = B.D1.idx[jB_ptr];
9      int jA_ptr = i * A.D1.ptr + j;
10     int kB_ptr = B.D2.ptr[jB_ptr];
11     int kc_ptr = c.D0.ptr[0];
12     while (kB_ptr < B.D2.ptr[(jB_ptr + 1)]
13            && kc_ptr < c.D0.ptr[1]) {
14       int kB = B.D2.idx[kB_ptr];
15       int kc = c.D0.idx[kc_ptr];
16       int k = min(kB, kc);
17       if (kB == k && kc == k) {
18         A.vals[jA_ptr] += B.vals[kB_ptr]
19                         * c.vals[kc_ptr];
20       }
21       if (kB == k) kB_ptr++;
22       if (kc == k) kc_ptr++;
23   }}}
```

(c) Sparse B_{ijk} and c_k, Dense A_{ij}

Figure 1: Generated code for $A_{ij} = \sum_k B_{ijk} * c_k$ with different data layouts of B and c.

- A *C++ Library* implementation of our compiler technique called `taco`, short for Tensor Algebra Compiler (§ 6).
- A demonstration of the performance of `taco`-generated code compared to hand-coded implementations from state-of-the-art widely used linear and tensor algebra libraries. We show that `taco` generates efficient code for both simple kernels like SpMV and complex kernels like the Matricized Tensor Times Khatri-Rao (MTTKRP) product (§ 7).

Our technique can be used in libraries such as TensorFlow [3], or Eigen [20] or integrated into the compilers of MATLAB [28], Julia [13] or Simit [23].

2. Motivation

It is well understood how to write high performance code for dense linear and tensor algebra operations. Such codes only require a single loop nest that accesses arrays using simple affine indices, and it is well known how to optimize such loops. However, sparse linear and tensor algebra and complex compound expressions are a completely different story. First, sparsity comes in many different formats. For example a sparse matrix, a 2-tensor, can be stored using many different formats such as Compressed Sparse Row (CSR) and Columns (CSC), DCSR/DCSC for also compressing the first dimension, or BCSR/BCSC/BDCSR which includes dense inner blocks. For higher-order tensors there are more formats, including Compressed Sparse Fiber (CSF). Having multiple sparse tensors in an operation causes a combinatorial blow-up of the number of kernels needed. Second, many important problems require compound epxressions like $B^T c + d$, $B \circ (CD)$ or $B_{(1)}(C \odot D)$. Forcing computation kernels to divide these into multiple simpler operations at the smallest granularity requires producing too many intermediate results, reducing available locality and hindering performance. Thus, it is important to compute the result using a loop nest that makes a single pass over the expression. This makes it impossible to create a library of optimized functions as there are too many to consider. Third, the code to simultaneously iterate over a dimension of multiple sparse structures is non-trivial. Consider a tensor-by-vector multiplication expression

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

where i, j and k are index variables ranging over the tensor dimensions. The code to evaluate this expression depends entirely on the formats of the three operands. The simplest code is when the formats are all dense row-major, as shown in Figure 1(a). It is simple to iterate over the $m \times n \times p$ iteration domain to compute the required values, since the input tensor and vector are dense.

However, if most entries in B are zero it is more efficient to only store the non-zeros, reducing the storage cost from $\theta(m \times n \times p)$ to $\theta(nnz)$, where nnz is the number of non-zeros. Sparse tensor representations such as Compressed Sparse Fiber (CSF) [37] do this, but the code to iterate through the non-zero subset of the iteration space is complicated. Figure 1(b) shows code for computing the expression when B is stored with CSF where all dimensions are compressed, while c and A remain dense. The loops iterate through the subset of each dimension in B that contains non-zeros (lines 1–13), performs a multiplication, and stores the result in the correct place in A (lines 18–19).

The code is even more complex when both operands are sparse. In Figure 1(c), the inner loop simultaneously iterates over the sparse dimensions of B and the sparse entries of c, and computes only when there is a non-zero entry in *both* operands with the same location in dimension k, as shown on line 17. This restriction makes the innermost loop tricky to implement, since it needs to run through the k dimension

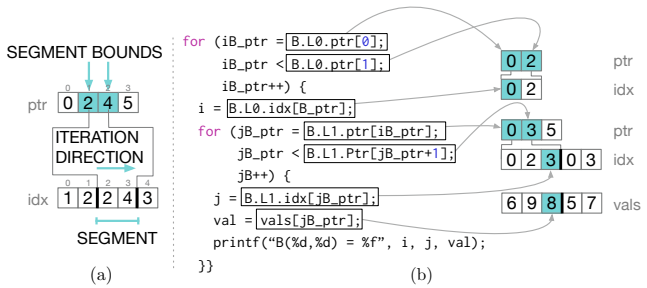


Figure 2: (a) Understanding iteration and access for a `Sparse` storage level. Segment bounds are given by values in the `ptr` array, while `idx` stores index values for the segment. (b) Iteration through a 2-tensor stored in ($\texttt{Sparse}_{d1}$,$\texttt{Sparse}_{d2}$) format to access the highlighted value, showing the correspondence between code and storage.

of both operands simultaneously (lines 12–16) and decide whether to compute a value at each loop iteration.

It is hard write optimized sparse tensor code by hand, since it depends on the tensor dimensionality, storage formats and the expression itself. In fact, the tensor-vector multiplication example in Figure 1 shows only three out of the 768 possible implementations that are needed to handle all the combinations of formats we support. Hand-coding 768 kernels for this expression alone is not realistic. The technique in this paper obviates the need to write this code, making it possible to mix and match formats to implement any tensor algebra operation automatically.

3. Tensor Storage

In this section, we describe a space of tensor formats that our technique supports. Many matrix and tensor formats have been proposed in the literature and several important ones are points in our space, such as CSR, CSC, BCSR, DCSR and CSF. However, tensors can have any order (dimensionality) so there are an unbounded number of formats. For it to be possible to develop a general code generation approach, we developed a way to describe formats recursively from simple composable parts that also lead to a composable code generation algorithm. Consider B from our running example, which is a tensor with three dimensions. The simplest storage format is a dense multidimensional array, but it is wasteful if most components are zeroes.

Intuitively, we find it convenient to think of a tensor as a tree with one tree level per dimension (plus a root node), as shown for a 2-tensor in Figure 3(a)-(c). In this formulation, each tree-path represents a tensor coordinate with a non-zero value. Each path starts from the root going down to a leaf. 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 dimensions of B are stored, the levels of the tree occur in different order, e.g. (d_1, d_2, d_3) or (d_3, d_1, d_2) where d_i is a dimension.

In our technique, the storage *format* of a tensor is given by the order dimensions are stored in, and for each dimension, whether it is stored using dense or sparse (compressed) level storage. Tensor values are always stored in a separate array, but the tensor format index arrays are necessary to interpret them. For each kind of level storage we store index metadata:

`Dense` requires only storing the size of the dimension, since it stores all indices in the dimension.

`Sparse` stores only the subset of the corresponding dimension that has non-zero values. This requires two index arrays, `ptr` 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 `ptr` array stores the location in the `idx` array where each segment begins. Thus segment i is stored in locations `ptr[i]:ptr[i+1]` in the `idx` array (there's a sentinel at the end of `ptr` 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 those in the CSR matrix format. In addition to higher-order tensors, our formulation lets us represent several common sparse matrix formats. Figure 3, shows all 8 ways to store a 2-tensor (matrix) using our technique. The first column shows dense row- and column-major storage. The second column shows the CSR and CSC formats. Both are represented as (`Dense`,`Sparse`), but the order dimensions is switched. The third column shows the (`Sparse`,`Dense`) format, which is not as commonly-used but is useful for some circumstances (see § 7). Finally, the fourth column shows ($\texttt{Sparse}_{d1}$,$\texttt{Sparse}_{d2}$) format, which corresponds to Doubly-Compressed Sparse Row (DCSR) [16]. We also support the corresponding column-first format DCSC. Furthermore, the number of formats increases exponentially as the tensor dimensionality increases (actually $2^d d!$), which makes making hand-coding intractable. Other important sparse formats we support include sparse vectors, blocked CSR (which we represent as a 4-tensor), and the CSF format for higher order tensors, which is sparse in every dimension [37].

`Sparse` storage for a dimension does not allow efficient random access to indices and values. However, `Sparse` is optimized for iteration in a specific order. Figure 2(a) shows how a `Sparse` level is accessed. The `ptr` array gives the bounds for each segment in the `idx` array; iterating over the indices in a segment is a unit-stride access. Figure 2(b) shows the correspondence between code and storage for iterating through a ($\texttt{Sparse}_{d1}$,$\texttt{Sparse}_{d2}$) 2-tensor for printing out the non-zero entries; the arrows show the current positions of each loop when printing the highlighted value. The complexity of iterating through `Sparse` levels is one of the reasons why writing sparse tensor code is difficult.

The two kinds of per-level storage we support can express a wide space of formats suitable for storing sparse tensors. We anticipate adding more kinds of per-level storage to support an even wider variety of formats in the future. Describ-

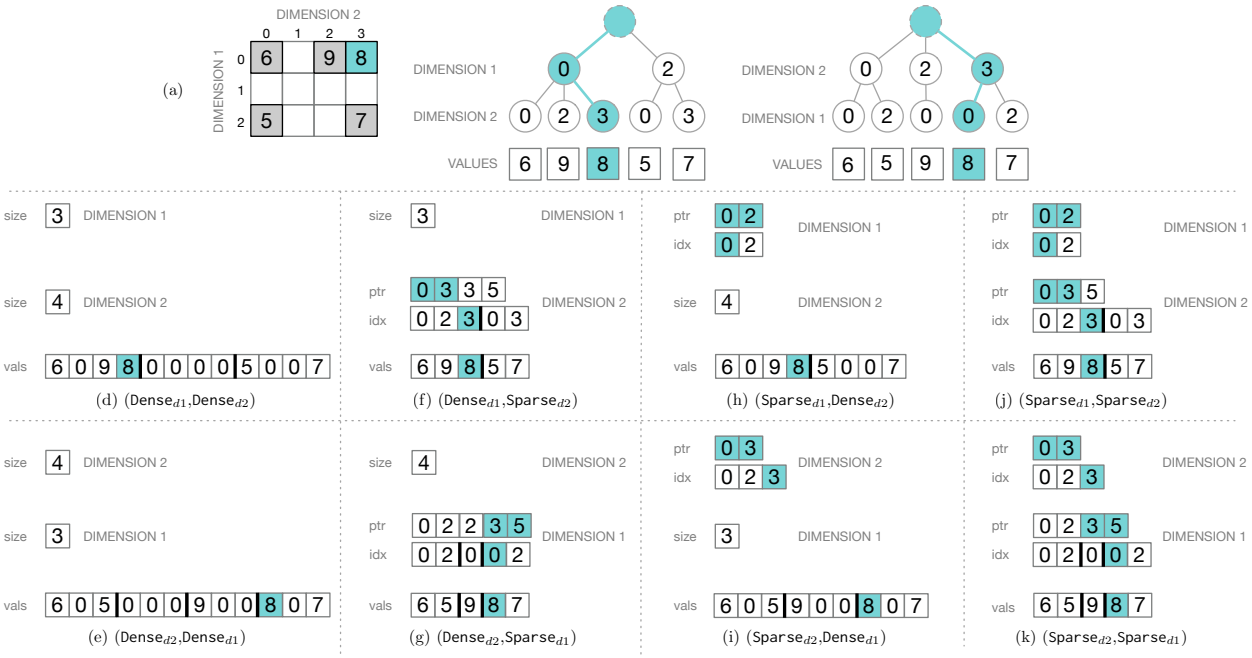


Figure 3: The matrix (2-tensor) is shown in (a) with the tree that describes how to access elements if dimension 1 is accessed before dimension 2, and reverse. The storage formats in (d) and (e) are row-major and column major dense matrix storage. (f) and (g) are similar to Compressed Sparse Row (CSR) and Column (CSC) formats which only stores nonzeros of the tensor. (h) stores full rows of the tensor, but omits empty rows while (i) does the same for columns. (j) and (k) are similar to Doubly-Compressed Sparse Row (DCSR) and Column (DCSC) formats, for storing hypersparse matrices used in graph algorithms [16].

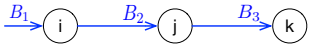
ing the space of tensor storage formats in this manner allows us to support an unbounded number of formats and to use a modular code generation approach that generates code specific to each tensor storage level, as described in § 5.

4. Iteration Schedules

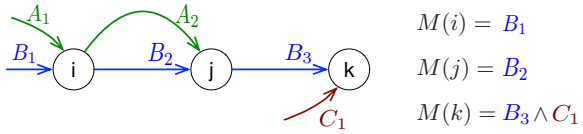
Iteration Schedules describe how to iterate over the non-zero values of a tensor expression and are the intermediate representation of our approach. They are sufficiently general to let us produce efficient code from any tensor expression, from SpMV to Matricized Tensor Times Khatri-Rao products ($A_{ij} = B_{ikl} * C_{kj} * D_{lj}$) and beyond. They also represent the access restrictions on sparse tensors, which makes them ideal for generating code from sparse expressions.

Tensor storage is recursively defined and specify each dimension to be either dense or sparse. Thus, the storage provide an order for iterating over the dimensions of the tensor. We call this order a *tensor path* and it is the key building block of iteration schedules. A tensor storage format can support multiple paths if some dimensions are dense. A sparse dimension supports only one path direction as that dimension needs to be iterated on before its children, while a dense dimension can be iterated through before or after its children. For example a dense row-major matrix can be iterated in (d_1, d_2) order or (d_2, d_1) order. However, a CSR matrix can only be efficiently iterated in (d_1, d_2) order.

To make this concrete, consider the tensor index expression from § 2 $A_{ij} = \sum_k B_{ijk} * c_k$ and the loops that iterates over the indices of sparse B in Figure 1(b). An iteration schedule is a graph where the index variables (i,j,k) of an expression become vertices and where read expressions (B_{ijk}) become directed tensor paths.



In this example, each index variable vertex corresponds to a loop nest in the code. Further, the tensor path of B_{ijk} corresponds to the loop bounds, which iterate over the per-dimension levels of the tensor storage to visit its non-zero values. In § 3 each path in a tensor storage tree corresponded to a non-zero tensor value. A tensor path symbolically represents all the paths in the forest of trees of operands. Furthermore, each tensor read expression results in a tensor path. In our example there are three read expressions:



In this example two operand paths meet at k. In Figure 1(c) we showed that if both B and c are sparse then special code is needed to merge their indices. This need to merge tensor indices is described in iteration schedules using *merge rules*. Every index variable has a merge rule and they are propositional logic expressions where the atoms are steps of tensor paths. The merge rule of k is $M(k) = B_3 \wedge c_1$, which means the code iterates over those values of k for

which both B and c have non-zero values. Finally, the path for A_{ij} in this schedule is special; it is the result path. Result paths do not take part in merge rules, since we iterate over the operand and not the result indices.

We are now ready to define iteration schedules.

DEFINITION 1. *An iteration schedule is a graph defined by the ordered tuple $S = (V, P, M(V))$ comprising a sequence $V = (i_1, i_2, ..., i_n)$ of index variable vertices, a set P of directed tensor paths through vertices, and a map M from each index variable to a merge rule.*

DEFINITION 2. *A tensor path $p \in P$ is an ordered tuple of o index variable vertices, where o is a positive number. The kth index variable p_k of tensor path is called its kth step.*

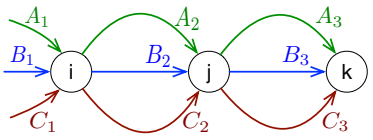
DEFINITION 3. *A merge rule is a propositional logic expression (no quantifiers) where the atoms are tensor path steps.*

Iteration schedules are constructed from index expressions as follows. The expression's index variables become the iteration schedule's vertices. Each sub-expression that reads a tensor value (e.g. B_{ijk}) results in a tensor path that has length o, where o is the tensor order. The order of the steps in the path is determined by the order of the levels in the tensor's storage. If the levels are ordered (d_1, d_2, d_3), where d_1 is the tensor's first dimension, then the path is (i, j, k). Conversely, if the levels are ordered (d_2, d_1, d_3) then the path is (j, i, k) and so on.

Next, the merge rules are constructed for each index variable in turn, by rewriting the index expression to propositional logic. First, we replace the operators that are annihilated by zero ($0 \otimes a = 0$), such as $*$ and $/$, with conjunctions ($\wedge$). Next, we replace operators that are not annihilated by zero ($a \oplus 0 = a$), such as $+$ and $-$, with disjunctions ($\vee$). Finally, we replace the index expression's tensor read operands with the corresponding tensor path step that is incoming on the index variable. If the tensor path is not incoming on this index variable then we throw the sub-expression away.

As a final step, we order the index variables. The ordering constraint is that the graphs formed by the tensor paths must be a directed acyclic graph (no cycles). Any topological order will do, so we order the index variables by a topological sort. If no topological order exists, then it is not possible to produce an iteration schedule, and the user must re-order a tensor's storage to remove cycles.

To make this concrete we will construct the iteration schedule for a tensor addition $A_{ijk} = B_{ijk} + C_{ijk}$.



$M(i) = B_1 \vee C_1$

$M(j) = B_2 \vee C_2$

$M(k) = B_3 \vee C_3$

The expression's index variables i, j and k are our vertices. Further, the expression has three tensor reads A_{ijk}, B_{ijk} and C_{ijk} and these become our tensor paths. Let's assume the storage for all three tensors are ordered by increasing dimension (d_1, d_2, d_3). If we name the paths by the tensor that is read, then we have $P = \{A, B, C\}$, $A = (i, j, k)$, $B = (i, j, k)$ and $C = (i, j, k)$. Finally, we must construct one merge rule for each index variable. Since the expression is an addition and since every tensor path goes through every index variable, all three merge rules are disjunctions: $M(i) = B_1 \vee C_1$, $M(j) = B_2 \vee C_2$ and $M(k) = B_3 \vee C_3$.

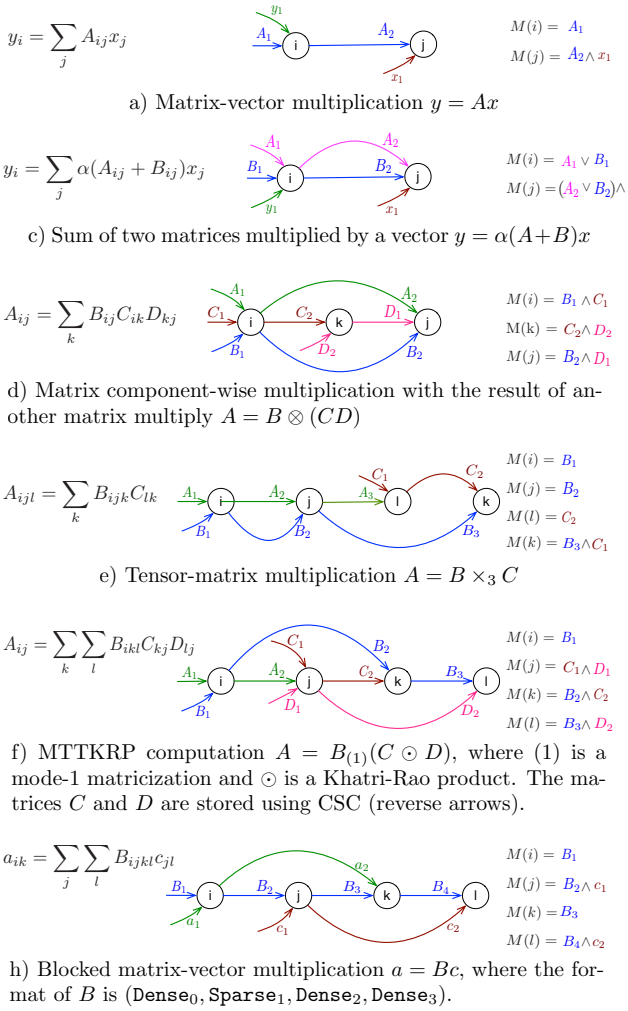


Figure 4: The index expression and the iteration schedule for a few selected matrix and tensor computations.

Figure 4 contains more examples of iteration schedules, ranging from a simple matrix-vector multiplication to blocked matrix-vector multiplication and MTTKRP.

5. Code Generation

The tensor storage formats from § 3 and the iteration schedules from § 4 come together in this section to generate loops that iterate over the sparse iteration space of an index expression. The challenge in code generation is three-fold. First, we must separate the code generation for different index variables so that we can compile arbitrarily-complex expressions from simple, composable building blocks. Second, we

```
1  int a1_ptr = a.d1.ptr[0];
2  int b1_ptr = b.d1.ptr[0];
3  int ic_ptr = c.d1.ptr[0];
4  while (b1_ptr < b.d1.ptr[1] && ic_ptr < c.d1.ptr[1]) {
5    int ib = b.d1.idx[b1_ptr];
6    int ic = c.d1.idx[ic_ptr];
7    int i = min(ib, ic);
8
9    if (ib == i && ic == i)
10     a.vals[a1_ptr++] = b.vals[b1_ptr] + c.vals[ic_ptr];
11   else if (ib == i)
12     a.vals[a1_ptr++] = b.vals[b1_ptr];
13   else if (ic == i)
14     a.vals[a1_ptr++] = c.vals[ic_ptr];
15
16   if (ib == i) b1_ptr++
17   if (ic == i) ic_ptr++;
18 }
19 while (b1_ptr < b.d1.ptr[1]) {
20   a.vals[a1_ptr++] = b.vals[b1_ptr++];
21 }
22 while (ic_ptr < c.d1.ptr[1]) {
23   a.vals[a1_ptr++] = c.vals[ic_ptr++];
24 }
```

Figure 5: C code for sparse vector addition $a_i = b_i + c_i$.

must generate code that merges the iteration spaces of tensors that can both be added and multiplied together. Third, we must insert compute and index assembly statements at the correct levels of the emitted loops. We will first introduce a new concept we call merge lattices that will help us generate merge code (§ 5.1). We will then present a general code generation algorithm (§ 5.2), before finally addressing how to insert compute and assembly statements (§ 5.3).

5.1 Merge Rules and Merge Lattices

As we described in § 4 each index variable has an associated merge rule. The merge rule of an index variable specifies how the tensor storage indices of incoming tensor path steps should be merged. Merge rules consist of two operators: conjunctions ($\wedge$) and disjunctions ($\vee$). A conjunction means the index variable iterates over the intersection of the incoming tensor path steps. A disjunction means the index variable iterates over the union of the incoming tensor path steps. A merge rule can merge any number of tensor path steps, so we will develop a general scheme that iterates over the set combination of any number of tensor path steps.

The motivation for merge lattices is that it is expensive to merge two sparse indices using a disjunction, because the merge loop must check whether each of the merged indices still have values. For this reason algorithms such as the two-finger merge algorithm with three loops to merge indices were developed. In a two-finger merge the first loop iterates until one index runs out of values, followed by two loops to merge in the rest of the index that still has values. We generalize this insight and introduce a new representation we call merge lattices that we will depend on to generate merged loops in § 5.

Let us first consider a concrete example. Sparse vector addition $a_i = b_i + c_i$ requires a disjunction merge as the non-zero values in a must be the union of the non-zero values of b and c. The reason for this is that addition is not annihilated

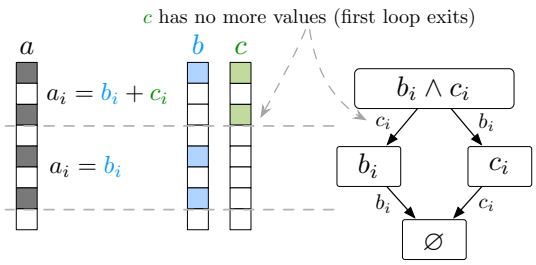


Figure 6: Sparse vector addition example $a_i = b_i + c_i$ and the merge lattice for i's disjunction merge rule $b_i \vee c_i$.

by zero ($a + 0 = a$), which means that if any of the operands of a scalar addition is non-zero then the result is non-zero. Figure 5 shows the C code to add two sparse vectors and the left hand side of Figure 6 shows an example. The code performs a two-finger merge that iterates over the union of the sparse operands using three loops. The first loop on line 4 iterates while both `a` and `b` have any remaining values. Line 7 in the loop body computes the value of `i` as the smallest of the index values of `b` and `c`. If both indices have a value at `i` they are added together on line 10. Otherwise the index with a value at `i` is stored in `a` at lines 12 and 14. Finally, lines 16–17 increment the `ptr` variable of the indices that had a value at `i`. After the first loop has terminated one of the vectors may have more values left. The loops at lines 19–24 iterates over the remaining values of `b` or `c` and stores them in `a`.

The disjunction code for vector addition has three loops. As we saw in Figure 1(c), a conjunction merge has one merge loop (the inner-most loop). In general an n-ary merge requires more loops. To describe these loops we introduce the concept of a merge lattice.

DEFINITION 4. *A merge lattice $\mathcal{L}$ is an ordered lattice, consisting of n ordered lattice points ($\mathcal{L}_1, ..., \mathcal{L}_n$). A lattice point $\mathcal{L}_p$ is a conjunctive merge of m indices associated with tensor path steps, and the lattice points are ordered on indices running out of values.*

Figure 6 depicts the merge lattice of the vector addition. The top of the lattice represents the first merge loop, which iterates while any of the merged indices have values left. The middle represents the additional loops that are needed to merge sub-expressions that still have values. Finally, the bottom of the merge lattice is when there are no more index variable values left to consider. The arrows (ordering) of the merge lattice represents a merged index running out of values, which means a while loop terminates and control moves on to the next loop.

Merge lattices rewrite merge rules to a canonical form containing a sequence of disjunctions where the terms are conjunctions. For example, a disjunction is re-written as

$$b_i \vee c_i = (b_i \wedge c_i) \vee (b_i) \vee (c_i)$$

Each disjunction becomes a while loop in a sequence of loops as we'll see in § 5.2. With this insight we can con-

struct a merge lattice for any merge rule as follows. Let us first define multiplication on merge lattice points, and both multiplication and addition on merge lattices.

DEFINITION 5. *Let the multiplication of two lattice points $\mathcal{L}_p \times \mathcal{L}_q$ be the concatenation of their tensor path steps.*

DEFINITION 6. *Let the multiplication of two merge lattices $\mathcal{L}^1 \times \mathcal{L}^2$ be the cartesian product of all their lattice points $(\mathcal{L}^1_0, ..., \mathcal{L}^1_n) \times (\mathcal{L}^2_0, ..., \mathcal{L}^2_m)$.*

DEFINITION 7. *Let the addition of two merge lattices $\mathcal{L}^1 + \mathcal{L}^2$ be their multiplication, followed by the lattice points in $\mathcal{L}^1$, followed by the lattice points in $\mathcal{L}^2$.*

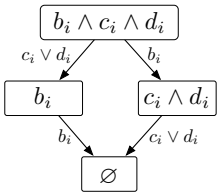
Given these operations we can recursively construct a merge lattice from a bottom up traversal of a merge rule using the following construction rules:

- *Tensor path step atom*: construct a merge lattice with one lattice point that contains the tensor path step.
- *Conjunction*: multiply sub-expression merge lattices.
- *Disjunction*: add sub-expression merge lattices.

To see this algorithm in action, consider the vector expression $a_i = b_i + (c_i * d_i)$; a combined addition and component-wise multiplication. The merge rule for i is $b_i \vee (c_i \wedge d_i)$. Starting at the expression leaves we construct a merge lattice for each of the operands containing one lattice point each. We then compute the merge lattice for $(c_i \wedge d_i)$ by multiplying the operand merge lattices, producing a merge lattice with just one lattice point $[[c_i \wedge b_i]]$. Finally, the top expression is a disjunction so we add the merge lattices for the sub-expressions

$$[[a_i]] + [[c_i \wedge d_i]] = [[a_i \wedge c_i \wedge d_i] \vee [a_i] \vee [c_i \wedge d_i]].$$

This merge lattice is shown here pictorially. The top represents a loop that iterates over the conjunction of all three indices. If either c_i or d_i are exhausted the lattice drops down to the lattice point for b_i. Furthermore, if b_i is exhausted the lattice drops down to the lattice point $[c_i \wedge d_i]$.



5.2 Code Generation Algorithm

In this section we describe how to generate code to compute a tensor index expression whose operands are a mix of dense and sparse tensors. Figure 7 contains the recursive algorithm that emits code that iterates over the merged iteration space of an index expression described by its iteration schedule. The algorithm uses a meta-programming syntax where compiler code is colored blue while emitted code is colored black and surrounded by quotation marks. In emitted code blue text denotes the value of compiler variables.

The algorithm generates code for one index variable vertex at a time. For each index variable it emits code to merge the iteration spaces of the indices of the incoming tensor paths, recursively emitting code for the next iteration variable where appropriate. To achieve this, the algorithm defines three mutually recursive functions that take an index variable as arguments: `codegen`, `merge-dense` and `merge-sparse`. The `codegen` function drives the recursion and produces code for an index variable using either `merge-dense` or `merge-sparse` (if there are any index variables left). The `merge-dense` function is used if the merge rule for the index variable only merges dense index dimensions. It produces dense loops that iterate over all entries in the tensor dimension corresponding to i_k. In the loop body it computes the sub-expressions that can be computed at that loop level (if any), and recursively calls `codegen` to generate code for the next index variable.

The `merge-sparse` function generates loops that merge sparse index dimensions. Figure 7(a) shows generated code for a matrix add, where the inner dimension is a sparse merge. Numbered parts shows code generated from different stages of code generation, and parts 4–5 shows the loops that merges sparse indices. The code generation relies heavily on the merge lattices from the previous section to generate this code. The body of a while loop first merges the index variables produced by each sparse index by taking the smallest value (parts 6). The intuition is that we step through the iteration space in increasing order, and the indices with the smallest index value are next.

The next step is to produce nested if statements that handle the various cases of the merge such as one index having a value at that point, all indices having a value, etc. (parts 8). Each if statement computes the sub-expression that can be computed for that case, recursively generate code for the next variable given the case expression and insert values into sparse output indices. See § 5.3 for information on how this is managed. Finally, the `merge-sparse function` emits code that conditionally increments those indices that were just used to compute (parts 9). The jB index value of B is compared to the min index value j. If they are equal, then we move to the next location in B.

5.3 Computation and Tensor Index Assembly

In the previous section we showed how to generate the loop nests. However, we left two functions undefined, namely `compute` and `insert`. These functions are called in the case handling in step 8 of Figure 7 at each recursive level. Note that these functions are independent and it is possible to generate code that uses either or both of them. This makes it possible to emit code that only assembles the indices, only computes, or both computes and assembles indices. In many iterative applications tensor values change, but not their structure and it is useful to be able to assemble the indices in the outer loop and then merely compute thereafter.

The `compute` function emits a scalar expression to compute the index expression in the base case of the loop nests. The operands of the expression are read from the operand

```
int B0_ptr = 0;
int C0_ptr = 0;
int A0_ptr = 0;
for (int i = 0; i < B.D0.ptr; i++) {
  int B1_ptr = (B0_ptr * B.D0.ptr) + i;
  int C1_ptr = (C0_ptr * C.D0.ptr) + i;
  int A1_ptr = (A0_ptr * A.D0.ptr) + i;

  int B2_ptr = B.D1.ptr[B1_ptr];
  int C2_ptr = C.D1.ptr[C1_ptr];
  while (B2_ptr < B.D1.ptr[B1_ptr+1] &&
         C2_ptr < C.D1.ptr[C1_ptr+1]) {
    int jB = B.D1.idx[B2_ptr];
    int jC = C.D1.idx[C2_ptr];
    int j = min(jB, jC);
    int A2_ptr = (A1_ptr * A.D1.ptr) + j;

    if (jB == j && jC == j)
      A.vals[A2_ptr] = B.vals[B2_ptr] + C.vals[C2_ptr];
    else if (jB == j)
      A.vals[A2_ptr] = B.vals[B2_ptr];
    else if (jC == j)
      A.vals[A2_ptr] = C.vals[C2_ptr];

    if (jB == j) B2_ptr++;
    if (jC == j) C2_ptr++;
  }

  while (B2_ptr < B.D1.ptr[B1_ptr+1]) {
    int jB = B.D1.idx[B2_ptr];
    int j = min(jB);
    int A2_ptr = (A1_ptr * A.D1.ptr) + j;

    if (jB == j)
      A.vals[A2_ptr] = B.vals[B2_ptr];

    if (jB == j) B2_ptr++;
  }

  while (C2_ptr < C.D1.ptr[C1_ptr+1]) {
    int jC = C.D1.idx[C2_ptr];
    int j = min(jC);
    int A2_ptr = (A1_ptr * A.D1.ptr) + j;

    if (jC == j)
      A.vals[A2_ptr] = C.vals[C2_ptr];

    if (jC == j) C2_ptr++;
  }
}
```

a) Sparse matrix add example ($A_{ij} = B_{ij} + Cij$), where B, C are CSR matrices (`Dense`, `Sparse`) and A is dense (`Dense`, `Sparse`). The algorithm in b) and c) has been applied to generate code. Following code generation, conditional constant and copy propagation can be applied to simplify the code, removing the conditional in the unary while loops.

```
codegen(index-expr, iteration-schedule)
  # emit code that sets the 0th ptr variables
  for I in indices of iteration-schedule
    emit "int I_0_ptr = 0;"
  end
  codegen(index-expr, i_1)
end

codegen(index-expr, i_k)
  if  k <= n
    if M(i_k) merges only dense tensor path steps
      merge-dense(index-expr, i_k)
    else
      merge-sparse(index-expr, i_k)
    end
  end
end
```

b) Top-level code generation functions

```
merge-dense(index-expr, i_k)
  let d be the size of the tensor dimension i_k iterates over
  emit "for (int i_k = 0; i_k < d; i_k++) {"
  for I_j in M(i_k)
    emit "int I_j_ptr = (I_j-1_ptr * d) + i_k;"
  end
    compute(index-expr, i_k)
    codegen(i_k+1)
  emit "}"
end


merge-sparse(index-expr, i_k)
  let L be the merge lattice of merge rule M(i_k)

  # emit code to initialize sparse ptr variables
  for I_j in M(i_k) if I_j is sparse
    emit "int I_j_ptr = I_j.ptr[I_j-1_ptr];"
  end

  for L_p in lattice points of L
    let sparse-indices = [I_j in L_p if I_j is sparse]
    let dense-indices  = [I_j in L_p if I_j is dense]

    # emit code to iterate while all the sparse indices have more values
    let c = "&&".join(["I_j_ptr < I_j.ptr[I_j_ptr+1]" for I_j in sparse-indices])
    emit "while(c) {"

    # emit code to compute sparse index variables
    for I_j in sparse-indices
      emit "int i_k_I_j = I_j.idx[I_j_ptr];"
    end

    # emit code to combine sparse index variables using min
    let index-variables = ", ".join(["i_k_I_j" for I_j in sparse-indices])
    emit   "int i_k = min(index-variables);"

    # emit code to compute dense ptr variables
    for I_j in dense-indices
      let d be the size of the tensor dimension i_k iterates over
      emit "int I_j_ptr = (I_j-1_ptr * d) + i_k;"
    end

    # emit code for each case of the merge lattice points dominated by L_p
    let ifcond = " && ".join(["i_k_I_j == i_k" for I_j in sparse-indices])
    emit   "if (ifcond) {"
             compute(index-expr, i_k)
             codegen(case-expr, i_k+1)
             insert(I_j, i_k)
    emit     "I_k_ptr++;" where I_k is the result index corresponding to k
    emit   "}"
    for L_q in lattice points strictly dominated by L_p in level order
      let elifcond = " && ".join(["i_k_I_j == i_k" for I_j in Lq if I_j is sparse])
      emit "else if (elifcond) {"
             let case-expr = sub-expression(expr, Lq)
             compute(case-expr, i_k)
             codegen(case-expr, i_k+1)
             insert(I_j, i_k)
             emit "I_k_ptr++;" where I_k is the result index corresponding to k
      emit "}"
    end

    # conditionally increment the sparse ptr variables
    for I_j in sparse-indices
      emit "if (i_k_I_j == i_k) I_j_ptr++;"
    end
    emit "}"
  end
end
```

c) Dense and sparse index variable merge code generation functions.

Figure 7: Code generation algorithm for tensor index notation expressions given an iteration schedule (§ 4). The generated code iterates over the combined sparse iteration space of the index expression, computes values and inserts them into the result tensor. The example code (a) and algorithm (b and c) are tagged with matching numbers.

vals arrays at the location of their last ptr variable, and the result is stored into the result vals array at the location of it's last ptr variable. For example,

```
A.vals[A2_ptr] = B.vals[B2_ptr] + C.vals[C2_ptr];
```

This is a simple solution. A more sophisticated scheme would insert expressions into the highest loop level where their last ptr variables are available.

The insert function takes care of building the index structure for sparse levels in the result. Recall that the index structure for sparse levels consists of two arrays: ptr that contains the beginning of each index segment, and idx which contains the index values for all the segments. The insert function sets the values of both. The idx array is set to the index location at the current level (i_k in the code generation algorithm) and the current ptr array location is incremented to reflect the additional value. For example,

```
A.d2.idx[A2_ptr++] = j;
A.d2.ptr[A1_ptr + 1] = A2_ptr;
```

However, there is a complication in outer loops. It can happen that the sub-computation did not produce any values. For example, in an elementwise matrix multiplication two rows might both have values, but the intersection might not. To prevent empty locations in the result index structure (legal, but sub-optimal compression) we emit code to check if the sub-computation produced non-zeroes. For example,

```
if (A.d2.ptr[A1_ptr+1] > A.d2.ptr[A1_ptr]) {
  A.d1.idx[A1_ptr++] = i;
  A.d1.ptr[A0_ptr + 1] = A1_ptr; }
```

The conditional tests whether the current and previous ptr for the sub-loops are the same. If they are not, the sub-loops produced values, so we insert a new location into the index.

Finally, it is necessary to allocate memory for the result tensor. This can be handled by emitting code to check whether there is more space left in the idx and ptr and vals arrays before they are written to. If there is no more space left, then the emitted code allocates more memory. We recommend doubling the memory when running out and to emit code to shrink it after the loop nest.

6. taco: The Tensor Algebra Compiler

We have implemented the technique in this paper in a C++ library called taco (short for Tensor Algebra Compiler). Figure 8 demonstrates how to compute the tensor-vector multiplication shown in § 2 using taco.

Tensor objects, which correspond to mathematical tensors, are created by specifying the dimensions, the type of its entries, and 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 § 3. On lines 1–6 in our example for instance, A is defined to be a 1024×1024 DCSC matrix of doubles, B is defined to be a $1024 \times 1024 \times 2048$ CSF tensor of doubles, and c is defined

```
1  Format dcsc({Sparse,Sparse}, {1,0});
2  Format csf({Sparse,Sparse,Sparse}, {1,0,2});
3  Format dv({Dense}, {0});
4  Tensor<double> A({1024,1024}, dcsc);
5  Tensor<double> B({1024,1024,2048}, csf);
6  Tensor<double> c({2048}, dv);
7
8  B.insert({0,0,0}, 1.0);
9  B.insert({1,2,0}, 2.0);
10 B.insert({1,2,1}, 3.0);
11 c.insert({0}, 4.0);
12 c.insert({1}, 5.0);
13 B.pack();
14 c.pack();
15
16 Var i, j, k(Var::Sum);
17 A(i,j) = B(i,j,k) * c(k);
18
19 A.compile();
20 A.assemble();
21 A.compute();
```

Figure 8: C++ tensor-vector multiplication using taco.

to be a dense vector of doubles of length 2048. Tensor that only serve as inputs to computations can be initialized with user-specified data as illustrated on lines 8–14.

Tensor algebra computations are expressed with tensor index notation, as shown on lines 16–17. Note the resemblance between line 17 and the mathematical expression of tensor-vector multiplication from the beginning of § 2. Var objects in taco correspond to indices in tensor index notation, with summation reductions implied over variables declared as type Sum (such as k in the example).

Once a tensor algebra computation is defined, invoking compile on the target of the computation (A) prompts taco to generate code that evaluates the computation. taco does this by applying the algorithm described in § 5. Next, the assemble method assembles the sparse index structure of the output tensor and preallocates its memory. Finally, the actual computation is performed by invoking the compute method to execute the code generated by compile.

Alternatively, we can invoke assembleCompute to simultaneously assemble index structures and compute values, which is the approach many libraries take. However, in many applications the matrix or tensor values change, but not their structure. Since allocating memory and assembling indices is expensive, it is beneficial to be able to separate these tasks.

7. Results

To demonstrate the performance of taco on the linear algebra subset of tensor algebra, we compare it to four widely used existing sparse linear algebra libraries: Eigen [20], pOSKI [46], uBLAS [47], and Gmm++ [34]. Eigen, uBLAS and Gmm++ are all examples of C++ libraries that exploit templates to specialize linear algebra operations for fast execution wherever 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 [3]. pOSKI is a C library that automatically

tunes sparse linear algebra kernels to take advantage of optimizations such as register blocking and vectorization.

A limitation of existing sparse linear and tensor algebra libraries is that the developers must write code for every combination of kernels and formats they wish to support. For this reason, they typically choose a subset of formats to support. However, real-world matrices and tensors benefit from different formats and thus kernels. To demonstrate this, in § 7.3 we show results for four classes of real-world matrices that benefit from different formats. In § 7.4 we demonstrate that some matrices benefit greatly from blocked storage, which further motivates a general compiler approach.

Finally, we demonstrate the performance of `taco` on tensor algebra by comparing to two existing sparse tensor algebra libraries, namely SPLATT [38] and the MATLAB Tensor Toolbox [8]. SPLATT is a high-performance C++ toolkit designed with sparse tensor factorization in mind. The MATLAB Tensor Toolbox is a more general library for MATLAB that also implements a number of sparse tensor factorization algorithms and also supports a variety of more primitive operations on general (non-factorized) sparse tensors.

7.1 Methodology

All of our experiments were run on a cluster of two-socket Intel Xeon E5-2695 v2 machines running at 2.4 GHz with 30 MB of L3 cache and 128 GB of main memory. The machines run Ubuntu 14.04.5 and all tests were compiled using GCC 5.4. Each run was made in exclusive mode, and we use 100 timing measurements to obtain our final results. Because `taco` currently does not support parallelism, all runs are done without parallelism.

We got inputs for our experiments from several sources. We obtained sparse matrices from real-world applications from the SuiteSparse Matrix Collection [17]. Sparse tensors were assembled from a data set of wall posts from the Facebook New Orleans networks [45] and the Enron email dataset [1]; the sparse tensor assembled from the Facebook data set has dimensions $1591 \times 63891 \times 63890$ and contains 737934 non-zero elements, while the sparse tensor assembled from the Enron data set has dimensions $86321 \times 184 \times 184$ and contains 125409 non-zero elements.

7.2 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication (SpMV) is one of the most important operations in sparse linear algebra, given its use in iterative methods for solving linear systems. We evaluated the performance of SpMV code generated by `taco` for matrices stored in the (`Dense`,`Sparse`) and (`Sparse`,`Sparse`) formats and compared it against that of SpMV kernels implemented in existing sparse linear algebra libraries.

The results of this experiment, shown in Figure 9, clearly demonstrate that the techniques described in § 5 that `taco` implements is indeed capable of generating efficient SpMV kernels that are at least competitive in terms of execution time with all existing libraries we compared against. For every input matrix, `taco` is able to generate a competitive SpMV kernel for at least one supported sparse matrix format. Note that we compare against running pOSKI without tuning here, which executes CSR SpMV; we compare against tuned (blocked) pOSKI in § 7.4.

7.3 Choice of Matrix Format

Existing sparse linear algebra libraries tend to support a limited set of sparse matrix formats. For instance, support for sparse matrix storage in Eigen is restricted to the CSC and CSR formats, while pOSKI supports only CSR and BCSR matrices. However, many real-world applications deal with matrices that have structures that make using variants of CSR less than ideal.

Figure 10 shows representatives of four classes of real-world matrices. For each matrix, we show results of matrix-

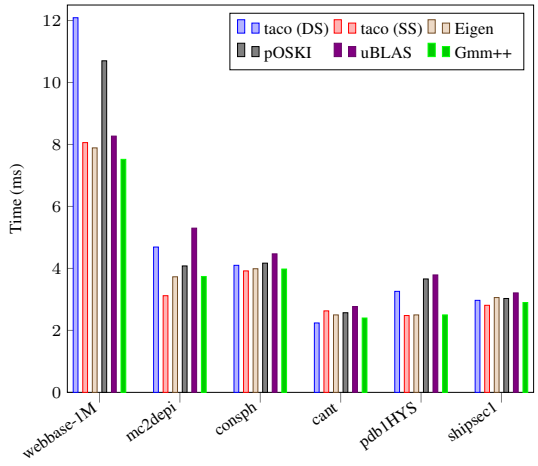
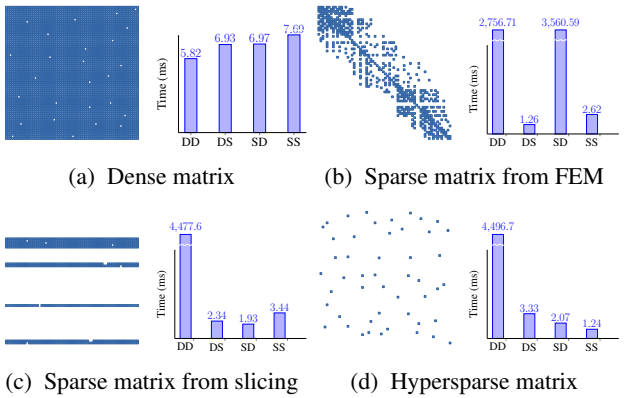


Figure 9: SpMV performance on matrices from real-world applications using `taco` and other existing libraries.



(a) Dense matrix (b) Sparse matrix from FEM

(c) Sparse matrix from slicing (d) Hypersparse matrix

Figure 10: Performance of SpMV 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 average execution time of SpMV using the formats listed on the axis to store the matrix.

vector multiplications with the matrix stored in each of four formats. The results show that each matrix benefits from a different format with the same properties as the matrix's sparsity pattern, which demonstrates the importance supporting multiple formats. Figure 10(a) shows a dense matrix, which benefits from (Dense,Dense) storage. Figure 10(b) shows the kind of sparse matrix that is ubiquitous in mesh code such as a finite element simulation or a geometry optimization problem, which tends to have a small bounded number of entries on each row. As mentioned above, these matrices perform well when stored using (Dense,Sparse) storage (CSR). Figure 10(c) show a matrix where most rows are empty, but where the non-empty rows are dense. These matrices can result from slicing a dense matrix and benefit from (Sparse,Dense) storage. Finally, Figure 10(d) shows a hypersparse matrix, which frequently show up in graph computations [16]. These matrices have many empty rows and non-empty rows have few values, which makes it inefficient to store either of the two dimensions using dense storage.

7.4 Block Matrices

Matrices that originate from physical domains often exhibit structure that are mostly sparse but contain small dense *blocks* of nonzeros. pOSKI takes advantage of such matrices by implementing optimized code for the Blocked Compressed Sparse Row (BCSR) format. In taco, the equivalent to this is a 4-tensor, where the inner tensor dimensions are stored as Dense.

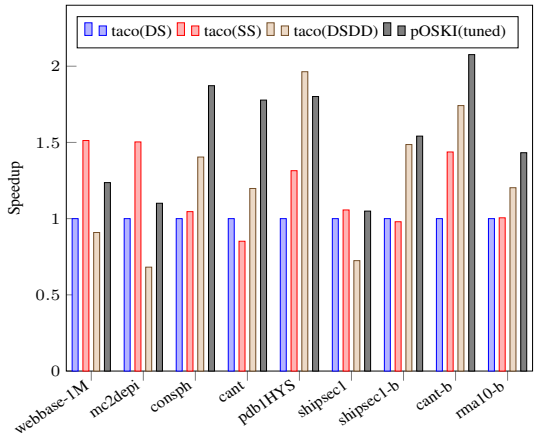


Figure 11: Performance of blocked SpMV on various matrices using taco compared with (tuned) pOSKI.

Figure 11 compares taco performance with pOSKI for the six matrices from § 7.2 and three synthetic matrices obtained by filling in 3×3 blocks inside an existing matrix (the rightmost -b matrices). We show speedup relative to taco (Dense,Sparse) format. In addition to the taco formats shown before, we show performance for tuned pOSKI and taco's BCSR-equivalent format. Tuned pOSKI uses auto-tuning plus a cost model to determine the best block size for the specific matrix; due to time constraints, we only compare against a single block size (3×3) for taco. For three of the matrices, one of the taco formats outperforms pOSKI; in two cases, the performance is nearly identical, and in four cases pOSKI has the highest performance. pOSKI shows that auto-tuning over a large number of block sizes is an effective way to speed up computation on some matrices; such auto-tuning can also be built on top of taco. Overall, these results show that even without auto-tuning, the wide variety of formats taco supports can result in higher overall performance, depending on matrix structure.

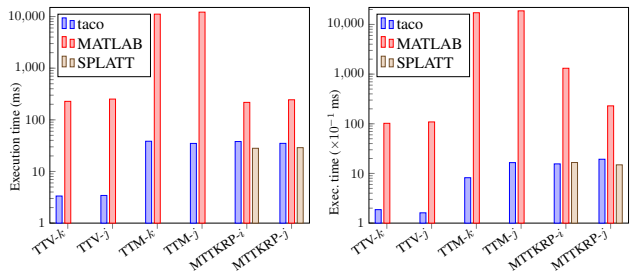
7.5 Tensor Algebra

We assessed the effectiveness of our technique for sparse tensor algebra by comparing the performance of several tensor algebra kernels generated by taco with the same kernels implemented in the MATLAB Tensor Toolbox and SPLATT. We focused our attention on the following set of kernels, all of which are commonly used in real-world applications:

1. $A_{ij} = \sum_k B_{ijk} * c_k$
2. $A_{ik} = \sum_j B_{ijk} * c_j$
3. $A_{ijl} = \sum_k B_{ijk} * C_{lk}$
4. $A_{ilk} = \sum_j B_{ijk} * C_{lj}$
5. $A_{il} = \sum_{j,k} B_{ijk} * C_{jl} * D_{kl}$
6. $A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$

The first two kernels correspond to mode-k and mode-j tensor-vector multiplications (TTV-k and TTV-j) while the third and fourth correspond to mode-k and mode-j tensor-matrix multiplications (TTM-k and TTM-j). The last two are called mode-i and mode-j matricized-tensor times Khatri-Rao products (MTTKRP-i and MTTKRP-j), which along with TTM form an essential part of many algorithms for computing tensor decompositions like the Tucker decomposition and the canonical polyadic decomposition [38].

We show the average time it takes to execute each of the kernels on sparse tensor inputs assembled from the Facebook and Enron datasets in Figure 12. (Note that taco uses the (Sparse,Sparse,Sparse) format to store its sparse tensor input for this experiment.) No results are shown for tensor-vector multiplication and tensor-matrix multiplication for SPLATT as those particular kernels are not implemented in the library, which already suggests the usefulness of a technique that can emit arbitrary kernels without requiring a library developer to manually implement it.



(a) Facebook dataset (b) Enron dataset

Figure 12: Performance of sparse tensor algebra kernels generated with taco and implemented in other existing libraries.

In all of the cases we examined, we observe that the performance of the `taco`-emitted kernel exceeded that of the equivalent MATLAB Tensor Toolbox kernel by a notable margin, usually an order of magnitude or more. We also observed that the performance of `taco`-emitted MTTKRP kernels was competitive with SPLATT's hand-optimized kernel in a sequential setting (even exceeding it for the Enron tensor when computing the mode-i MTTKRP). This demonstrates that our technique is able to generate high-performance sparse tensor algebra codes that can be used in real-world applications.

8. Related Work

There are several lines of prior work on libraries, languages and compilers for dense and sparse linear and tensor algebra.

Dense Linear and Tensor Algebra There has been a lot of work on languages [13, 22, 28], libraries [2, 5, 20, 36, 43, 48], and compilers [31, 39] for dense linear algebra and loop transformations that can optimize dense loop nests [29, 49, 50]. The Tensor Contraction Engine [7] is a framework for automated optimization of dense tensor contractions developed for the quantum chemistry simulation software NWChem. TensorFlow is a recent interface for machine learning algorithms that passes dense tensors between kernels in a dataflow computation [3]. However, this work does not directly carry over to sparse linear algebra compilation due to complications introduced by indirect references.

Sparse Linear Algebra Libraries and Languages The use of general sparse matrices goes back to Tinney and Walker [41] and an early library for sparse matrix operations is described by McNamee [30]. Gustafson [21] later expanded these operations to include matrix-matrix multiplication. More recently MATLAB [28], Julia [13], Eigen [20] and PETSc [11] have become popular for computing with sparse matrices. MATLAB and Eigen are general systems that support all basic linear algebra operations. However, their sparse matrix formats are limited to coordinate, CSR and CSC. PETSc targets distributed systems and scientific computing. Another well known library is OSKI [46] (and the parallel pOSKI), developed to explore auto-tuning of select sparse kernels. However, the feature set is limited to SpMV, Tridiagonal Solves, Matrix powers, and simultaneously multiplying a matrix and its transpose by vectors.

Sparse Linear Algebra Compilers Most related to our approach is previous work on compiling sparse linear algebra. Several researchers have presented techniques to compile dense linear algebra loops to sparse linear algebra loops. Bik and Wijshoff [14, 15] developed a compiler framework that compiled dense loops computing on dense arrays, where zeros in the arrays make the computation a no-op, to sparse loops over the non-zero values of those arrays. They used a technique they call guard encapsulation to move non-zero guards into sparse data structures.

The Bernoulli project [25–27, 40] reduced declarative constraint expressions that enumerate sparse iteration spaces to relational algebra queries, converting sparsity guards into predicates on relational selection and join expressions. This avoided having to find a sequence of loop transformations that result in the right form for guard encapsulation. They then build on techniques from the database literature to optimize queries and insert efficient join implementations. The Bernoulli approach to code generation is less general than ours as they only support conjunctive loops with unary merges. For example, SpMV and the linear combination of rows version of SpMM [25, Introduction and Chapter 5]. They conjecture that their approach can be extended to disjunctive binary loops (e.g. matrix addition) by implementing binary disjunctions as outer joins [40, Chapter 15.1], but they did not explore this further [25, Chapter 11.2].

Venkat et al. present the transformations *compact* and *compact-and-pad* which turn dense loops with a conditional guard into into loops over a sparse matrix in one of several formats [44]. However, they do not discuss loops with more than one sparse matrix, which require merging indices and gives rise to many different schedules.

SIPR [33] is an IR for sparse matrix operations that generates calls to a C++ library to implement sparse code from dense code. SIPR can handle row swaps, but does not address index merging in a general way, restricting what kinds of element-wise operations are possible. Further, LL [6] is a small language designed for functional verification of sparse formats and can generate code for binary sparse operations as well as verifying their correctness. However, LL does not generate code for compound linear algebra.

More recently, Sparso demonstrated that context can be exploited to optimize sparse linear algebra programs by reordering matrices and taking advantage of matrix properties [35]. These optimizations are orthogonal to our technique and can reinforce each other.

In contrast, our approach generalizes beyond linear algebra to sparse tensor expressions, while supporting compound linear algebra expressions. In addition, we start with index expressions instead of loops, freeing us from needing to derive programmer intent from arbitrary code.

Sparse Tensor Algebra An early system for sparse tensor computations is the MATLAB Tensor Toolbox [9].The tensor toolbox provides several hand-coded kernels for computing important tensor operations using coordinate format. SPLATT is an optimized C library with support for shared memory parallelism [38]. It supports fast MTTKRP operations and tensor contractions. Finally, TensorFlow recently added some support for sparse tensor computations in the form of hand-coded kernels [3]. However, this work appears to still be in its infancy as the supported operations are limited. In contrast, our approach is to compile kernels as opposed to hand-coding them.

9. Conclusions and Future Work

We have presented the first technique to compile any compound sparse linear and tensor algebra expression to efficient loops that make one pass over the expression's sparse iteration space. This puts sparse linear and tensor algebra on a firm compiler foundation to build on. Implemented in a library or behind a linear algebra language it lets the programming system shape code around the data structures at hand, so that data does not need to be converted. We believe code should be malleable, so that data can be at rest.

We see four main directions for future work. First, we will open source `taco` so that it can be used directly or incorporated into full fledged linear algebra languages such as Julia and Simit. Second, we believe that our tensor storage technique can be extended to support other important formats such as coordinate, ellpack and dia. Third, we plan on implementing the ability to traverse sparse formats in reverse directions, which can be useful if the data you have does not match the ideal layout for the computation at hand. We also believe our approach can be extended to support parallel, distributed and accelerator architectures and for the first time provide true portability for this class of problems.

References

[1] `http://cis.jhu.edu/ parky/Enron/enron.html`.

[2] Intel math kernel library reference manual. Technical report, 630813-051US, 2012. `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf`.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[4] A. Anandkumar, R. Ge, D. Hsu, S. M. Kakade, and M. Telgarsky. Tensor decompositions for learning latent variable models. *J. Mach. Learn. Res.*, 15 (1):2773–2832, Jan. 2014. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=2627435.2697055`.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. ISBN 0-89871-447-8 (paperback).

[6] G. Arnold. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes*. PhD thesis, University of California, Berkeley, 2011.

[7] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[8] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007. doi: 10.1137/060676489.

[9] B. W. Bader and T. G. Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[10] B. W. Bader, M. W. Berry, and M. Browne. *Discussion Tracking in Enron Email Using PARAFAC*, pages 147–163. Springer London, London, 2008. ISBN 978-1-84800-046-9.

[11] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[12] J. Bennett and S. Lanning. The netflix prize. In *KDD Cup and Workshop 2007*, August 2007.

[13] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, September 2012. URL `http://arxiv.org/abs/1209.5145`.

[14] A. J. Bik and H. A. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*, pages 416–424. ACM, 1993.

[15] A. J. Bik and H. A. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing*, pages 57–75. Springer, 1994.

[16] A. Buluc and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS).*, pages 1–11, April 2008. doi: 10.1109/IPDPS.2008.4536313.

[17] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL `http://doi.acm.org/10.1145/2049662.2049663`.

[18] T. Delmarcelle and L. Hesselink. The topology of symmetric, second-order tensor fields. In *Proceedings of the Conference on Visualization '94*, VIS '94, pages 140–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-7803-2521-4. URL `http://dl.acm.org/citation.cfm?id=951087.951115`.

[19] R. Feynman, R. B. Leighton, and M. L. Sands. *The Feynman Lectures on Physics*. Addison-Wesley, 1963. 3 volumes.

[20] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[21] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, Sept. 1978. ISSN 0098-3500.

[22] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[23] F. Kjolstad, S. Kamil, J. Ragan-Kelley, D. I. Levin, S. Sueda, D. Chen, E. Vouga, D. M. Kaufman, G. Kanwar, W. Matusik, and S. Amarasinghe. Simit: A language for physical simulation. *ACM Transactions on Graphics*, 2015. To appear.

[24] J. C. Kolecki. An Introduction to Tensors for Students of Physics and Engineering. *Unixenguaedu*, 7(September):29, 2002. ISSN 18733514. doi: 10.1049/sqj.1936.0070.

[25] V. Kotlyar. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs*. PhD thesis, Cornell University, 1999.

[26] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*, pages 318–327. Springer, 1997.

[27] V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel sparse code for user-defined data structures. Technical report, Cornell University, 1997.

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

[29] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[30] J. M. McNamee. Algorithm 408: a sparse matrix package (part i)[f4]. *Communications of the ACM*, 14(4):265–273, 1971.

[31] T. Nelson, G. Belter, J. G. Siek, E. Jessup, and B. Norris. Reliable generation of high-performance matrix algebra. *ACM Trans. Math. Softw.*, 41(3):18:1–18:27, June 2015.

[32] C. NVIDIA. Cusparse library. *NVIDIA Corporation, Santa Clara, California*, 2014.

[33] W. Pugh and T. Shpeisman. Sipr: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing*, pages 213–229. Springer, 1999.

[34] Y. Renard. Gmm++. URL `http://download.gna.org/getfem/html/homepage/gmm/first-step.html`.

[35] H. Rong, J. Park, L. Xiang, T. A. Anderson, and M. Smelyanskiy. Sparso: Context-driven optimizations of sparse linear algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 247–259, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4121-9. doi: 10.1145/2967938.2967943. URL `http://doi.acm.org/10.1145/2967938.2967943`.

[36] C. Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical report, NICTA, Sept. 2010.

[37] S. Smith and G. Karypis. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 5. ACM, 2015.

[38] S. Smith, N. Ravindran, N. Sidiropoulos, and G. Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 61–70, May 2015. doi: 10.1109/IPDPS.2015.27.

[39] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 23. ACM, 2014.

[40] P. Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, 1997.

[41] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[42] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010. ISSN 0010-4655.

[43] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[44] A. Venkat, M. Hall, and M. Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, 2015.

[45] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[46] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005. ISSN 1742-6596. doi: 10.1088/1742-6596/16/1/071. URL `http://dx.doi.org/10.1088/1742-6596/16/1/071`.

[47] J. Walter and M. Koch. uBLAS. URL `http://www.boost.org/libs/numeric/ublas/doc/index.htm`.

[48] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[49] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, May 1991. ISSN 0362-1340. doi: 10.1145/113446.113449. URL `http://doi.acm.org/10.1145/113446.113449`.

[50] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. AAI8303027.