Tensor Algebra Compilation with Workspaces

Fredrik Kjolstad
MIT, USA
fred@csail.mit.edu

Peter Ahrens
MIT, USA
pahrens@csail.mit.edu

Shoaib Kamil
Adobe Research, USA
kamil@adobe.com

Saman Amarasinghe
MIT, USA
saman@csail.mit.edu

***Abstract*—This paper shows how to extend sparse tensor algebra compilers to introduce temporary tensors called workspaces to avoid inefficient sparse data structures accesses. We develop an intermediate representation (IR) for tensor operations called concrete index notation that specifies when sub-computations occur and where they are stored. We then describe the workspace transformation in this IR, how to programmatically invoke it, and how the IR is compiled to sparse code. Finally, we show how the transformation can be used to optimize sparse tensor kernels, including sparse matrix multiplication, sparse tensor addition, and the matricized tensor times Khatri-Rao product (MTTKRP).**

Our results show that the workspace transformation brings the performance of these kernels on par with hand-optimized implementations. For example, we improve the performance of MTTKRP with dense output by up to 35%, and enable generating sparse matrix multiplication and MTTKRP with sparse output, neither of which were supported by prior tensor algebra compilers.

***Index Terms*—sparse tensor algebra, concrete index notation, code optimization, temporaries, workspaces**

I. Introduction

Temporary scalar variables are important for optimizing loops that iterate over dense multi-dimensional arrays and sparse compressed data structures that represent tensors. Variables are cheap to access because they do not require address calculations, can be stored in registers, and can also be used to precompute loop-invariant expressions. Temporaries can also be higher-dimensional tensors that we call workspaces. Workspaces of low dimensionality (e.g., a vector) are cheaper to access than tensors of higher dimensionality due to simpler address calculation and better locality. This makes them profitable in loops that repeatedly access a tensor slice, and they can also be used to precompute loop-invariant higher-dimensional tensor expressions.

Workspaces provide additional opportunities for optimizing loops that compute on sparse tensors. Sparse tensors contain mostly zeros and are therefore stored in compressed irregular data structures. Dense workspaces can drastically reduce access and insertion cost when they substitute compressed tensors due to asymptotically faster random access and insertion. (The time complexity of random access into a compressed tensor is $O(\log n)$ from search and the insertion complexity is $O(n)$ from data movement.) Furthermore, simultaneous iteration over compressed data structures, common in sparse tensor codes, requires merge loops with many conditionals. By introducing dense tensor workspaces of low dimensionality to keep memory costs minimal, we can reduce the cost of insertion and replace merge loops with random accesses.

Prior work on sparse tensor compilation describes how to optimize sparse imperative code [1]–[3] and how to generate sparse code from high-level tensor index notation [4], [5]. It does not, however, consider optimizations that introduce temporary tensors. These are important in many sparse tensor kernels, such as tensor additions, sparse matrix multiplication (where all matrices are sparse) [6], and the matricized tensor times Khatri-Rao product (MTTKRP) used to factorize sparse tensors [7]. Without compiler support for workspaces we leave performance on the table. In fact, the sparse matrix multiplication kernel is asymptotically slower without workspaces [6].

This paper presents a compiler transformation that introduces temporary tensor workspaces into sparse code generated from tensor index notation by our prior work, taco [4]. This workspace transformation is expressed in a new intermediate representation (IR) called concrete index notation, which precisely describes when computations occur and where results are stored. The workspace transformation is programmatically invoked through a scheduling API, giving users control of when to apply it. We outline heuristics for invoking the transformation, but leave as future work a policy system to determine full schedules for sparse tensor algebra expressions. This policy system can be built on top of our scheduling API.

Our contributions are:

Concrete Index Notation We introduce a new tensor algebra IR that specifies loop order and temporary variables (Section IV).

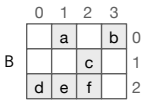
Workspace Transformation We describe a tensor algebra compiler transformation that can be used to remove expensive inserts into sparse tensors, eliminate merge code, and hoist loop invariant code (Section V).

Compilation We build on our prior work [4] to compile concrete index notation to sparse code (Section VI).

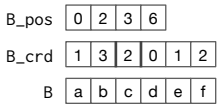
Case Studies We show how the transformation recreates sparse matrix multiplication, sparse matrix addition, and MTTKRP algorithms from the literature, while generalizing to new kernels.

We evaluate these contributions by demonstrating performance improvements from the use of workspaces, by observing that some kernels obtain asymptotic performance improvements, and by showing that the performance of resulting sparse code is competitive with hand-optimized implementations in the Intel MKL [8], Eigen [9], and SPLATT [7] high-performance libraries, including speedups of 4× over Eigen and 1.28× over MKL for sparse matrix multiplication.

978-1-7281-1436-1/19/$31.00 © 2019 IEEE



(a) Dense $m \times o$ matrix B



(b) Sparse CSR index of B

```
1  memset(A, 0, A1_size * A2_size * sizeof(float));
2  for (int i = 0; i < m; i++) {
3    for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
4      int k = B_crd[pB];
5      for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++)
6        int j = C_crd[pC];
7        A[i*n+j] += B[pB] * C[pC];
8      }
9    }
10 }
```

(c) $A_{ij} = \sum_k B_{ik}C_{kj}$ (sparse B, C)

```
1  memset(row, 0, row_size * sizeof(float));
2  for (int i = 0; i < m; i++) {
3    for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
4      int k = B_crd[pB];
5      for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
6        int jp = C_crd[pC];
7        row[jp] += B[pB] * C[pC];
8      }
9    }
10
11   for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {
12     int jc = A_crd[pA];
13     A[pA] = row[jc];
14     row[jc] = 0.0;
15   }
16 }
```

(d) $A_{ij} = \sum_k B_{ik}C_{kj}$ (sparse A, B, C)

Fig. 1: Figure (a) shows a sparse matrix B and (b) its compressed data structure. Figure (c) shows the code for a sparse matrix multiplication with a dense result and (d) with a sparse result. Sparse matrices do not support $O(1)$ insert so (d) uses a dense row workspace. The index data structures of the result matrix A have been pre-assembled and all memory pre-allocated.

II. Sparse Matrix Multiplication Example

We use matrix multiplication to introduce compressed data structures, sparse code and workspaces. These concepts, however, generalize to higher-dimensional tensor code. Matrix multiplication in index notation is

$$A_{ij} = \sum_k B_{ik}C_{kj}.$$

Sparse kernel code also depends on the tensor storage formats. Many formats exist and can be classified as dense formats (which store every component) or sparse/compressed formats (which store only nonzeros). Figure 1 shows two sparse matrix multiplication kernels: (c) with a dense result and (d) with a compressed sparse row (CSR) result matrix.

The CSR format and its column-major CSC sibling are ubiquitous in sparse linear algebra libraries due to their generality and performance [8]–[10]. In the CSR format, each matrix row is compressed (only nonzeros are stored). This requires two index arrays to describe the coordinates and positions of the nonzeros. Figure 1a shows a sparse matrix B and Figure 1b its CSR data structure. It consists of two index arrays B_pos and B_crd and a value array B. The array B_crd contains the column coordinates of the nonzero values in corresponding positions in B. The array B_pos stores the position in B_crd of the first column coordinate of each row, with a sentinel storing the number of nonzeros (nnz) in the matrix. Thus, contiguous values in B_pos store the beginning and end [inclusive-exclusive) of a row in the arrays B_crd and B. For example, the column coordinates of the third row are stored in B_crd at positions [B_pos[2], B_pos[3]). Finally, many libraries store the coordinates of each row in sorted order to improve the performance of some algorithms.

In Figure 1, both kernels multiply matrices using the *linear combination of rows* algorithm that computes a row of A as a sum of rows from C scaled by a row from B. When the matrices are sparse, the linear combinations of rows matrix multiply is preferable to inner products matrix multiply (where one result component is computed at a time) for two reasons. First, the sparse linear combinations are asymptotically faster because inner products must simultaneously iterate over row/column pairs and consider values that are nonzero in only one matrix [6]. Second, linear combinations of rows works on row-major matrices (CSR), while inner products require the second matrix to be column-major (CSC). It is often more convenient, as a practical matter, to keep matrices ordered the same way.

The matrix multiplication kernel in Figure 1c has two sparse CSR operand matrices but a dense result. Because it contains the subexpression B_{ik} it iterates over B's sparse matrix data structure with the loops over i (line 2) and k (lines 3–4). The loop over i is dense because the CSR format stores every row, while the loop over k is sparse because each row is compressed. To iterate over the column coordinates of the ith row, the k loop iterates over [B_pos[i], B_pos[i+1]) in B_crd.

The kernel in Figure 1d has a sparse CSR result to save memory when many of the values are zeros. The sparse result matrix complicates the kernel because the assignment on line 7 is nested inside the reduction loop k. This causes the inner loop j to iterate over and insert into each row of A several times. Sparse data structures, however, do not support fast random inserts (only appends). Inserting into the middle of a CSR matrix costs $O(\text{nnz})$ because the new value must be inserted into the middle of an array. To get the $O(1)$ insertion cost of dense formats, the kernel introduces a dense row workspace.

A workspace is a temporary tensor that is typically dense to get fast insertion and random access, with lower dimensionality than the result to keep down memory usage. Because values can be scattered efficiently into a dense workspace, the loop nest k, j (lines 3–9) in Figure 1d looks similar to the kernel in Figure 1c. Instead of storing values into the result matrix A, however, it stores them in a dense workspace vector. When a row of the result is fully computed in the workspace, it is appended to A in a second loop over j (lines 11–15). This loop iterates over the row in A's sparse index structure and thus assumes A's CSR index has been pre-assembled. Pre-assembling indices, common in simulation codes, increases performance when assembly can be moved out of inner loops,

```
1 // Create three square CSR matrices
2 Type mat(Float,{N,N});
3 Format CSR({Dense, Compressed});
4 TensorVar A(mat,CSR), B(mat,CSR), C(mat,CSR);
5
6 // Compute a sparse matrix multiplication
7 IndexVar i, j, k;
8 IndexExpr mul = B(i, k) * C(k, j);
9 IndexStmt matmul = (A(i, j) = sum(k, mul));
10
11 // Reorder to linear combinations of rows
12 matmul.reorder(k, j);
13
14 // Precompute the mul expression into a row workspace
15 IndexVar jc, jp;
16 Type vec(Float,{N});
17 TensorVar row(vec, Format({dense}));
18 matmul.precompute(mul, {{j,jc,jp}}, row);
```

Fig. 2: C++ code that constructs a matrix multiplication and transforms it with reorder and precompute scheduling commands into a linear combination of rows matrix multiplication with a dense workspace shown in Figure 1d.

but it is also possible to simultaneously assemble and compute. Section VI describes code that assembles result indices by tracking the nonzero coordinates inserted into the workspace.

III. Precompute Scheduling

We propose a scheduling extension to taco [4] in the spirit of Halide [11] to support programmatically reordering loops and precomputing subexpressions into workspaces. The reorder and precompute scheduling methods invoke the workspace transformation (Section V) and reorder transformation (Section IV-B). The C++ declaration of precompute is:

```
void
IndexStmt::precompute(IndexExpr expr,
                      vector<tuple<IndexVar,IndexVar,IndexVar>> vars,
                      TensorVar workspace);
```

We apply the precompute method to the index notation statement and provide as arguments the expression to precompute, the index variables to precompute over, and the workspace in which to store precomputed results. The expr argument is an index expression contained in the index statement on which we call precompute. The vars argument is a vector of index variable triplets of the form (old, consumer, producer). The old index variable is the variable we wish to precompute over and is split into the consumer and producer index variables. The producer variable is used to iterate over the statement that calculates the workspace, and the consumer variable is used to iterate over the statement that uses the workspace. Finally, the workspace argument is a tensor for storing the precomputed results and its order and dimensions must be large enough to contain all temporary values.

Figure 2 shows a C++ example that uses the reorder and precompute methods to transform an index notation statement so that it can be lowered to the code in Figure 1d. Lines 2–4 create three square CSR matrices with single precision floating point components, A, B and C. Lines 7–9 create a matrix multiplication expression using the taco index notation API. The initial order of the loops is ijk (free index variables first as described in Section VI), forming an inner product matrix multiplication algorithm. The matrices are stored in the CSR row-major matrix format, however, so the ijk loop order with the C_{kj} access causes the C matrix to be accessed in the column-major direction. Column-major access of a row-major sparse matrix is asymptotically inefficient. Line 12 therefore reorders k and j, making the access to C row-major and forming an ikj linear combination of rows matrix multiplication algorithm. The reorder introduces a new inefficiency. Since the j index variable used in A_{ij} is now inside the k summation index variable, the code will scatter values into A. The format of A is, however, row-major CSR which does not support efficient random insert. Lines 15–18 therefore precomputes and scatters the results into a dense row workspace that is then copied to A.

Dense workspaces are useful because they support $O(1)$ random access and insert. A workspace, however, can be any format including compressed and hash maps [5], [12]. Hash maps are particularly interesting, since they also support $O(1)$ random access and insert without the need to store all the zeros. Furthermore, the component type of a workspace can be different than the operand and result tensors, making mixed precision algorithms available in taco. For example, the row workspace in Figure 2 can be constructed with double precision floating point components to accumulate values in higher precision.

IV. Concrete Index Notation

Index notation is a popular input language for tensor algebra code generators and frameworks [4], [13], [14]. It describes what tensor operations should do, independently of how they are computed and how tensors are accessed. Thus, optimization decisions are not mixed with algorithmic descriptions.

While index notation is a suitable input language for tensor operations, it is unsuitable as a compiler IR. The reason is that it does not specify the order of execution or temporary tensors and their formats. Several existing representations could be used to fully describe how an index expression is computed, such as the low-level C code that implements the index expression, sparse extensions of the polyhedral model [15], [16], or iteration graphs [4]. These representations, however, are too unconstrained to conveniently apply the workspace transformations described in this paper.

We propose a new intermediate language for tensor operations called concrete index notation. Concrete index notation extends index notation with constructs that describe the order of loops, where to use temporaries and the formats of temporaries, without the details of how to coiterate over sparse data structures. In the compiler software stack, concrete index notation is an intermediate representation between index notation and low level imperative IR (see Figure 6). A benefit of this design is that we can transform concrete index notation instead of transforming sparse imperative code with indirect accesses, conditionals and while loops. The sparse imperative code is then generated when lowering concrete index notation.

statement :=	assignment	access :=	$\text{tensor}_{\text{indices}}$
	forall	indices :=	index$*$
	where	expr :=	literal
	sequence		access
assignment :=	access $=$ expr		(expr)
	access $+=$ expr ...		expr $+$ expr
forall :=	$\forall_{\text{index}}$ statement		expr expr
where :=	statement **where** statement		...
sequence :=	statement **;** statement		

Fig. 3: The grammar of concrete index notation. The construct $\forall_{i\ldots k}$ can be used as shorthand for $\forall_i \ldots \forall_k$ and we show only some of the incrementing assignments and binary expressions.

A. Statements

Concrete index notation has four statement types shown in Figure 3. The assignment statement assigns the result of an expression to a tensor component, the forall statement executes a statement over a range inferred from tensor dimensions, the where statement creates temporaries that store subexpressions, and the sequence statement reuses temporaries.

To describe concrete index notation, we return to the linear combinations of rows matrix multiplication example. We express the algorithm with forall loops in ikj order. Concrete index notation is on the left and corresponding loop nest pseudocode appears in gray on the right:

$$\forall_{ikj}\ A_{ij} \mathrel{+}= B_{ik}C_{kj}$$

```
A = 0
for i ∈ I
  for k ∈ K
    for j ∈ J
      A_ij += B_ik * C_kj
```

An **assignment** statement modifies the value of a single tensor component. It can either be a regular assignment ($=$) or an incrementing assignment (e.g., $+=$ but other operators are also allowed). In the matrix multiplication example, the incrementing assignment statement $A_{ij} \mathrel{+}= B_{ik}C_{kj}$ modifies the value of the tensor component A_{ij}. The statement is restricted so that the tensor on the left hand side may not appear on the right hand side of the expression and each tensor is accessed using only index variables. Finally, result tensors are implicitly initialized to zero.

A **forall** statement expresses iteration by binding an index variable to a set of values. Nesting of forall statements describes the order index variables are iterated in. In the matrix multiplication example, the three forall statements $\forall_i \forall_k \forall_j$, abbreviated as $\forall_{ikj}$, specify iteration order.

A **where** statement precomputes an expression into a temporary tensor variable. The matrix multiplication algorithm repeatedly computes and adds scaled rows to the matrix A. If A is stored in a segmented compressed data structure it is expensive to insert into rows, and we can use a where statement to accumulate into a dense workspace:

$$\forall_i \left(\forall_j\ A_{ij} = w_j\right) \textbf{ where } \left(\forall_{kj}\ w_j \mathrel{+}= B_{ik}C_{kj}\right)$$

```
for i ∈ I
  w = 0
  for k ∈ K
    for j ∈ J
      w_j += B_ik * C_kj
  for j ∈ J
    A_ij = w_j
```

The where statement introduces a dense temporary vector w to hold the partial sums of rows on the right-hand producer side, which is then copied to A on the left-hand consumer side.

A **sequence** statement allows tensor updates. Suppose we wish to add a sparse CSR matrix D to our multiplied matrices B and C. This operation ($A = D + BC$) is best accomplished by reusing our dense temporary vector w. We use the **sequence** (**;**) statement to denote result update:

$$\forall_i \left(\forall_j\ A_{ij} = w_j\right) \textbf{ where } \Big(\left(\forall_j\ w_j = D_{ij}\right) \textbf{;}\ \left(\forall_{kj}\ w_j \mathrel{+}= B_{ik}C_{kj}\right)\Big)$$

```
for i ∈ I
  w = 0
  for j ∈ J
    w_j = D_ij
  for k ∈ K
    for j ∈ J
      w_j += B_ik * C_kj
  for j ∈ J
    A_ij = w_j
```

Unlike a where statement, tensors defined on the left-hand side of a sequence statement can be modified on the right-hand side. They can therefore be used to express a sequence of tensor writes.

B. Reorder Transformation

Reordering concrete index notation statements is useful for several reasons. First, sparse tensors are sensitive to the order in which they are accessed. For example, iterating over rows of a CSC matrix is costly so we can reorder forall statements to yield better access patterns. We may also wish to reorder to move loop-invariant where statements out of inner loops. Critically, we may need to reorder statements so that the preconditions for our workspace transformation apply. When we reorder a concrete index statement, we want to know that it will do the same thing as it used to. We can ensure semantic equivalence by breaking larger reorder operations down into smaller equivalences. In all cases, we require that all the statements being reordered do not contain sequence statements.

Exchanging forall statements requires the associative property. If S is a statement which modifies its tensor with an assignment statement or an increment statement with an associative operator, then $\forall_i\ \forall_j\ S$ and $\forall_j\ \forall_i\ S$ are semantically equivalent.

Moving a forall into the consumer side of a where statement is similar to loop invariant code motion. If S_2 does not use the index variable j, then $(\forall_j\ S_1)$ **where** S_2 and $\forall_j\ (S_1$ **where** $S_2)$ are semantically equivalent.

Moving a forall into both sides of a where statement changes the reuse distance of the data. If the statement S_2 modifies its tensor with an assignment statement, then $(\forall_j\ S_1)$ **where** $(\forall_j\ S_2)$ and $\forall_j\ (S_1$ **where** $S_2)$ are semantically equivalent.

We can rearrange nested where statements when all consumers do not use all of the producers's tensors. If S_1 does not use the tensor modified by S_3, then $(S_1$ **where** $S_2)$ **where** S_3, and S_1 **where** $(S_2$ **where** $S_3)$ are semantically equivalent. If S_2 does not use the tensor modified by S_3 and S_3 does not use the tensor modified by S_2, then $(S_1$ **where** $S_2)$ **where** S_3, and $(S_1$ **where** $S_3)$ **where** S_2 are semantically equivalent.

V. Workspace Transformation

The workspace transformation precomputes tensor algebra subexpressions in a temporary workspace with the concrete index notation **where** statement. It can be used to optimize sparse tensor algebra kernels in the following ways:

Simplify merges Code to simultaneously iterate over sparse tensors contains expensive conditionals and loops. It can be simplified by precomputing subexpressions into dense workspaces (e.g., Figure 4).

Avoid expensive inserts Repeated accumulation into the middle of a sparse tensor is expensive. We can improve performance by adding results to a workspace with fast inserts, such as a dense array (e.g., Figure 1).

Hoist loop invariant code Computing everything in the innermost loop can result in redundant computation. Precomputing a subexpression in a separate loop and storing the results in a workspace can hoist parts of an inner loop (e.g., Figure 9).

The transformation rewrites concrete index notation to extract a subexpression and compute it separately, storing the results in a temporary tensor (the workspace). The subexpression can then be replaced by the temporary tensor in the main expression. Surrounding forall statements are pushed down into either the subexpression or the main expression to avoid redundant computation. The effect is that the original statement is split in two; one statement produces values for the other through the workspace. We can think of the workspace transformation as the multidimensional generalization of loop-invariant code motion, where we use temporary tensors instead of scalars to store loop-invariant values.

A. Algorithm

Let (S, E, I, w) be the inputs to the workspace transformation, where S is a statement of the form $\forall_J A_K \oplus= E \otimes F$ that does not contain sequences, I, J, and K are sets of index variables, and w is a workspace tensor. Let S' be the statement $A_K \oplus= E \otimes F$ in S. The transformation is requested and arguments are given through the precompute scheduling method shown in Section III. The workspace transformation then rewrites the statement to precompute E in a workspace as follows:

Let w be a fresh tensor variable.
Replace S' with $(A_K \oplus= w_I)$ **where** $(w_I \oplus= E)$.
for j in J, from innermost to outermost $\forall$ **do**
  Note that S' is of the form S'_C **where** S'_P.
  if j is used in both S'_C and S'_P and $j \in I$ **then**
    Replace $\forall_j S'$ with $(\forall_j S'_C)$ **where** $(\forall_j S'_P)$
  else if j is used only in S'_C **then**
    Replace $\forall_j S'$ with $(\forall_j S'_C)$ **where** S'_P
  else if j is used only in S'_P **then**
    Replace $\forall_j S'$ with S'_C **where** $(\forall_j S'_P)$
  else
    Error.
  end if
end for

The transformation can be applied to any binary operators $\otimes$ and $\oplus$ as long as neither has side effects and $\otimes$ distributes over $\oplus$. In order to apply the transformation, we may need to first rearrange the statement to match the form $\forall_J A_K \oplus= E \otimes F$. Note that if we have an operator $\odot(x, y) = x$, then we can use this operator to transform the expressions $A_K \oplus= E$ and $A_K = E$ to the required form. After executing the algorithm we can remove $\odot$ to simplify the resulting statements. We can also transform $A_K \oplus= w_I$ to $A_K = w_I$ when K contains I, meaning that we only increment each element of A once.

Depending on the forall statements containing S, the requested transformation may not be possible. The user needs to first reorder to remove forall statements over index variables that are unused in S' or index variables that are used in both S'_P and S'_C but not included in I. Alternatively, the user can add index variables to I, provided the dimensionality of resulting workspace w is the number of index variables in I and the dimension sizes are equal to the ranges of those index variables.

We show the workspace transformation is correct by examining each statement replacement. The first replacement (splitting the statement) is a scalar statement transformation to an equivalent form. The for loop contains the next three replacements. In the case where we move a forall statement into both S'_C and S'_P, we have checked that $j \in I$ and so this transformation moves from using the values of w_I immediately after assignment to using them after the j loop. The case where we move forall statements into the S'_C is equivalent to loop invariant code motion, and the case where we move a forall statement into S'_P is valid since $\otimes$ must distribute over $\oplus$.

Figure 4 shows the code before and after applying the workspace transformation to B_{ij} over j in a statement that computes the inner products of rows from two matrices. In this example the transformation replaces the while loop over j, which simultaneously iterates over the two rows, with a for loop that independently iterates over each row. The for loops have fewer conditionals, at the cost of reduced data locality. Note that sparse code generation is handled below the concrete index notation in the compiler stack, as described in Section VI.

B. Result Reuse Optimization

When applying a workspace transformation it may pay to reuse result tensors instead of introducing a new workspace. To support reuse, we use the sequence statement, which allows us to define a result and compute it in stages. Result reuse is useful in sparse vector addition with a dense result, as the partial results can be efficiently accumulated into the result:

$$\forall_i\ a_i = b_i + c_i \implies \big(\forall_i\ a_i = b_i\ ;\ \forall_i\ a_i \mathrel{+}= c_i\big).$$

That is, b is first assigned to a, and then c is added.

The workspace transformation can reuse the result as a workspace if the forall statements on both sides of the resulting sequence statement are the same; that is, the transformation does not hoist computation out of a loop. This precondition ensures that results do not get overwritten by their use as a workspace. For example, this precondition is not satisfied by the second transformation to the MTTKRP kernel in Section VII.

```
for (int i = 0; i < m; i++) {
  a[i] = 0;
  int pB2 = B2_pos[i];
  int pC2 = C2_pos[i];
  while (pB2 < B2_pos[i+1] && pC2 < C2_pos[i+1]) {
    int jB = B2_crd[pB2];
    int jC = C2_crd[pC2];
    int j = min(jB, jC);
    if (jB == j && jC == j) {
      a[i] += B[pB2] * C[pC2];
    }
    if (jB == j) pB2++;
    if (jC == j) pC2++;
  }
}
```

(a) Before: $\forall_{ij}\ a_i \mathrel{+}= B_{ij}C_{ij}$

```
for (int i = 0; i < m; i++) {
  a[i] = 0;
  memset(w, 0, w1_size * sizeof(double));

  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_crd[pB2];
    w[j] = B[pB2];
  }

  for (int pC2 = C2_pos[i]; pC2 < C2_pos[i+1]; pC2++) {
    int j = C2_crd[pC2];
    a[i] += w[j] * C[pC2];
  }
}
```

(b) After: $\forall_i\ (\forall_j\ a_i \mathrel{+}= w_j C_{ij})$ **where** $(\forall_j\ w_j = B_{ij})$

Fig. 4: Kernels that compute the inner product of each pair of rows in two CSR matrices ($a_i = \sum_j B_{ij}C_{ij}$) before and after applying the workspace transformation to the matrix B. The workspace transformation introduces a where statement that replaces the merge loop in (a) with two for loops in (b).

Figure 5 shows a sparse matrix addition with CSR matrices before and after applying the workspace transformation twice, resulting in a kernel with three loops. The first two loops add each of the operands B and C to the workspace, and the third loop copies the non-zeros from the workspace to the result A. The first workspace transformation applies to the subexpression $B_{ij} + C_{ij}$ over j resulting in

$$\forall_i\ (\forall_j\ A_{ij} = w_j)\ \textbf{where}\ (\forall_j\ w_j = B_{ij} + C_{ij})\,.$$

The second transformation applies to the B_{ij} subexpression on the producer side of the where statement. Without result reuse the result would be

$$\forall_i\ \big(\forall_j\ A_{ij} = w_j\big)\ \textbf{where}\ \Big(\big(\forall_j\ w_j = v_j + C_{ij}\big)\ \textbf{where}\ \big(\forall_j\ v_j = B_{ij}\big)\Big),$$

but with result reuse the two operands are added to the same workspace in a sequence statement

$$\forall_i\ (\forall_j\ A_{ij} = w_j)\ \textbf{where}\ (\forall_j\ w_j = B_{ij}\ \textbf{;}\ \forall_j\ w_j \mathrel{+}= C_{ij})\,.$$

C. Policy Heuristics

The three motivating goals of the workspace transformation can be used as simple heuristics that describe when kernels might benefit from the workspace transformation. Let I and i be sets of index variables where I contains i. Informally:

```
int pA2 = 0;
for (int i = 0; i < m; i++) {
  int pB2 = B2_pos[i];
  int pC2 = C2_pos[i];
  while (pB2 < B2_pos[i+1] && pC2 < C2_pos[i+1]) {
    int jB = B2_crd[pB2];
    int jC = C2_crd[pC2];
    int j = min(jB, jC);
    if (jB == j && jC == j) {
      A[pA2++] = B[pB2] + C[pC2];
    }
    else if (jB == j) {
      A[pA2++] = B[pB2];
    }
    else {
      A[pA2++] = C[pC2];
    }
    if (jB == j) pB2++;
    if (jC == j) pC2++;
  }
  while (pB2 < B2_pos[i+1]) {
    A[pA2++] = B[pB2++];
  }
  while (pC2 < C2_pos[i+1]) {
    A[pA2++] = C[pC2++];
  }
}
```

(a) Before: $\forall_{ij}\ A_{ij} = B_{ij} + C_{ij}$

```
memset(w, 0, w1_size * sizeof(double));
for (int i = 0; i < m; i++) {
  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_crd[pB2];
    w[j] = B[pB2];
  }

  for (int pC2 = C2_pos[i]; pC2 < C2_pos[i+1]; pC2++) {
    int j = C2_crd[pC2];
    w[j] += C[pC2];
  }

  for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
   int j = A2_crd[pA2];
   A[pA2] = w[j];
   w[j] = 0.0;
  }
}
```

(b) After: $\forall_i\ (\forall_j\ A_{ij} = w_j)$ **where** $(\forall_j\ w_j = B_{ij}\ \textbf{;}\ \forall_j\ w_j \mathrel{+}= C_{ij})$

Fig. 5: Kernels that add CSR matrices ($A_{ij} = B_{ij} + C_{ij}$) before and after applying the workspace transformation twice. The first application is to the expression $B_{ij} + C_{ij}$ over j. The second is to B_{ij} over j and reuses the workspace. The transformations replace the inner merge loop with two loops to accumulate the operands into w and one to copy w to A.

Simplify merges Statements of the form $(\forall_I A_i \oplus\!= B_I \otimes C_I \otimes D_I \otimes ...)$ merging more than 3 sparse tensors $B, C, D, ...$ into a sparse A result in expensive merges. We avoid this cost by creating a dense workspace, producing $(\forall_I A_i = w_i)$ **where** $(\forall_I w_i \oplus\!= B_I \otimes C_I \otimes D_I \otimes ...)$.

Avoid expensive inserts Statements of the form $(\forall_I A_i \oplus\!= E_I)$ where A is sparse will accumulate into a sparse output. Avoid sparse inserts by introducing a workspace to produce $(\forall_I A_I = w_I)$ **where** $(\forall_I w_I \oplus\!= E_I)$.

Hoist loop invariant code Statements of the form $(\forall_I A_I \oplus\!= E_I \otimes F_i)$ redundantly compute the expression F_i. Transform to make $(\forall_I \oplus\!= E_I \otimes w_i)$ **where** $(\forall_i w_i = F_i)$.

Applying these heuristics may require rewriting the expression first, and different equivalent forms of an expression may result in expressions with different performance characteristics.

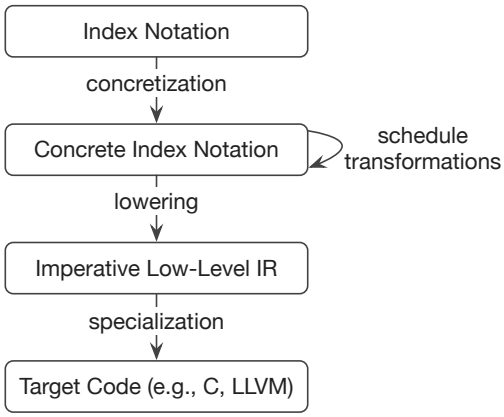


Fig. 6: Compiler stages from index notation, through concrete index notation with reorder and workspace schedule transformations, to low-level imperative IR and target code.

```
1  memset(A, 0, A1_size * A2_size * sizeof(double));
2  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
3    int i = B1_crd[pB1];
4    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
5      int j = B2_crd[pB2];
6      int pA2 = (i * A2_size) + j;
7      int pB3 = B3_pos[pB2];
8      int pc1 = c1_pos[0];
9      while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
10       int kB = B3_crd[pB3];
11       int kc = c1_crd[pc1];
12       int k = min(kB, kc);
13       if (kB == k && kc == k) {
14         A[pA2] += B[pB3] * c[pc1];
15       }
16       if (kB == k) pB3++;
17       if (kc == k) pc1++;
18     }
19   }
20 }
```

Fig. 7: Sparse generated code of a tensor-vector multiplication $\forall_{ijk}\ A_{ij} \mathrel{+}= B_{ijk}c_k$, where B and c are sparse.

VI. Compilation

This section describes the transformations to and from concrete index notation, called **concretization** and **lowering**. Figure 6 shows the compiler workflow, with IRs as boxes and IR transformations as arrows. Most transformations translate from an IR at a higher level of abstraction to an IR at a lower level of abstraction. The workspace transformation, however, transforms concrete index notation. As shown in Section III, these transformations are programmatically requested, by the user or by a policy system.

The concretization IR transformation transforms index notation into concrete index notation in two steps:

Insert forall statements for the index variables in the index notation expression. The forall statements of free index variables are nested outside those of reduction variables.

Replace reduce expressions with where statements whose producer substatement reduces into a scalar variable.

The result is valid concrete index notation that can be optimized with the workspace transformation and further lowered to low-level sparse imperative code.

The IR lowering transforms concrete index notation into sparse imperative C-like code. This algorithm internally uses the merge lattices we described in prior work but is simpler than the code generation algorithm described there [4]. The reason for the increased simplicity is that it lowers concrete index notation statements instead of iteration graphs. It therefore does not need to deduce at each recursive step what subexpressions are available to be emitted.

The code generation algorithm recurs on concrete index notation statements. When it encounters assignment statements it emits them as scalar code. When it encounters where statements the algorithm emits the producer side followed by the consumer side. Finally, when it encounters sequence statement it emits the left-hand side followed by the right-hand side.

The main complexity of sparse code generation is isolated to the code generation of forall statements, as these must simultaneously coiterate over hierarchical tensor data structures. We refer the reader to our prior work for more intuition about hierarchical coiteration and the iteration graph graphical notation [4, Section 4]. In the code generation approach we describe in this paper, iteration graphs have been replaced with concrete index notation. To generate code for a forall statement, the new code generation algorithm traverses the forall's body to collect all tensor modes that are indexed by the forall's index variable. The generated code for the forall statement must coiterate over these tensor modes by coiterating over their sparse data structures. For example, Figure 7 shows the sparse code to compute a sparse tensor-vector multiplication. The outer loops generated for the forall statements of i (lines 2–3) and j (line 4–5) iterate over only the sparse data structures of B, since i and j are only used to access B. Note that A is a result and does not need to be iterated over since its iteration space is the same as B by definition. The inner while loop generated for the forall statement k (lines 9–12), however, coiterates over the sparse data structures of the last mode of B and the vector c. This coiteration while loop iterates over their intersection since B and c are multiplied. If they were instead added then three while loops would be generated to coiterate over their union.

Forall statements are lowered to imperative code using merge lattices to generate merge code as described in our prior work [4, Section 5]. We refer the reader to this paper for a full explanation and limit this exposition to outlining the differences in the algorithm due to the use of concrete index notation. The recursive calls to generate code for concrete index notation substatements, however, are handled differently. When recursively generating code at a lattice point, the data structures that are exhausted at that point are collected and the concrete index notation substatement is rewritten to remove them by symbolically setting them to zero. Thus, we only recur on the parts of the statement that are not exhausted, which simplifies the algorithm.

In code listings that compute sparse results, we have so far shown only kernels that compute results without assembling

```
1  A_pos = malloc((m+1) * sizeof(int));
2  A_crd = malloc(A_crd_size * sizeof(int));
3
4  row = malloc(row_size * sizeof(bool));
5  memset(row, 0, row_size * sizeof(bool));
6  rowlist = malloc(row_size);
7
8  A_pos[0] = 0;
9  for (int i = 0; i < m; i++) {
10   rowlist_size = 0;
11   for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
12     int k = B_crd[pB];
13     for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
14       int j = C_crd[pC];
15       if (!row[j]) {
16         rowlist[rowlist_size++] = j;
17         row[j] = true;
18       }
19     }
20   }
21
22   // Sort row indices
23   sort(rowlist, rowlist_size);
24
25   // Make sure A_crd is large enough
26   if (A_crd_size < (A_pos[i] + rowlist_size)) {
27     A_crd_size = (A_pos[i] + rowlist_size) * 2;
28     A_crd = realloc(A_crd, A_crd_size * sizeof(int));
29   }
30
31   // Copy row workspace indices to A_crd
32   for (int prow = 0; prow < rowlist_size; prow++) {
33     int j = rowlist[prow];
34     A_crd[A_pos[i] + prow] = j;
35     row[j] = false;
36   }
37   A_pos[i+1] = A_pos[i] + rowlist_size;
38 }
39 A = malloc(A_pos[m] * sizeof(float));
```

Fig. 8: Sparse matrix multiply assembly kernel (the compute kernel is in Figure 1d). The coordinates of row i are inserted into rowlist on line 16 and copied to A on line 34. The array row guards against redundant inserts.

sparse index structures (Figures 1d, 5b, and 10). This let us focus on the loop structures without the added complexity of workspace assembly. Moreover, it is common in numerical code to separate the kernel that assembles index structures (often called symbolic computation) from the kernel that computes values (numeric computation) [6], [17]. The code generation algorithm we outlined in prior work [4] and modified here can emit either, or a kernel that simultaneously assembles the result index structures and computes its values.

When generating assembly kernels from concrete index notation, a workspace results in two arrays that together track the workspace nonzero coordinates. The first array (e.g., rowlist) is a list of coordinates that have been inserted into the workspace, and the second array (e.g., row) is a Boolean array that guards against redundant inserts into the coordinate list.

Figure 8 shows assembly code for the linear combination of rows sparse matrix multiplication example from Section II. It is generated from the same concrete index notation statement as the compute kernel in Figure 1d, so the loop structure is the same except for the loop to copy the workspace to A starting on line 32. In compute kernels, the index structure of A has been pre-assembled so the code generation algorithm emits a loop to iterate over A. In an assembly kernel, however, it emits code to iterate over the index structure of the workspace. Furthermore, the assembly kernel inserts into the workspace index (rowlist) on lines 15–18, instead of computing a result, and sorts the index list on line 23 so that the new row of A is ordered. Note that the sort is optional and only needed if the result must be ordered. Finally, the assembly kernel allocates additional coordinate memory on lines 26–29 by repeated doubling.

VII. Case Study: Matricized Tensor Times Khatri-Rao Product (MTTKRP)

The matricized tensor times Khatri-Rao product (MTTKRP) is the critical kernel in the alternating least squares (ALS) algorithm for computing the canonical polyadic decomposition of tensors [18]. This decomposition generalizes the singular value decomposition to higher-order tensors and has applications in data analytics [19], machine learning [20], neuroscience [21], image classification and compression [22], and other fields [23].

The ALS algorithm and the MTTKRP kernel can be used to factorize tensors of any order. The MTTKRP kernel for a k-order factorization consists of a k-order tensor multiplied by $k-1$ matrices in different modes. For example, the 4-order MTTKRP kernel is $A_{ij} = \sum_{klm} B_{iklm} C_{mj} D_{lj} E_{kj}$. In this section we apply the workspace transformation twice to optimize the 3-order MTTKRP kernel; however, equivalent transformations apply to higher-order kernels. The first workspace transformation in this section results in an MTTKRP kernel roughly equivalent to the hand-optimized implementation in SPLATT [7], while the second transformation enables MTTKRP with sparse matrices.

The MTTKRP kernel for factorizing 3-order tensors is expressed in tensor index notation as

$$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$$

and multiplies a 3-order tensor by two matrices in the l and k dimensions. This requires four nested loops: the three outermost loops iterate over the sparse data structure of B, while the innermost loop iterates over the full range of j.

A concrete index notation statement for MTTKRP is

$$\forall_{iklj}\ A_{ij} \mathrel{+}= B_{ikl} C_{lj} D_{kj}.$$

In this statement, the forall statements have been reordered from the initial concretized form, so the sparse tensor B is traversed in the order of its sparse hierarchical data structure. When we apply the workspace transformation to $B_{ikl}C_{lj}$ at j the statement is transformed to precompute the subexpression in the workspace w, resulting in the statement

$$\forall_{ik}\ (\forall_j\ A_{ij} \mathrel{+}= w_j D_{kj})\ \textbf{where}\ (\forall_{lj}\ w_j \mathrel{+}= B_{ikl} C_{lj})\,.$$

The workspace transformation has split the forall statement of the j index variable into two forall statements, one on each side of the where statement. Notice that the index variable l is not used to index any tensors on the consumer (left) side of the where statement. As a result, the l forall statement has been eliminated from that side. The loop elimination was our purpose in applying the transformation as it caused the $w_j D_{kj}$ multiplication to be computed at a higher level in the resulting

```
 1  memset(A, 0, A1_size * A2_size * sizeof(double));
 2  memset(w, 0, w1_size * sizeof(double));
 3  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
 4    int i = B1_crd[pB1];
 5    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
 6      int k = B2_crd[pB2];
 7      for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
 8        int l = B3_crd[pB3];
 9        for (int j = 0; j < n; j++) {
10          int pC2 = (l * C2_size) + j;
-11         int pD2 = (k * D2_size) + j;
-12         int pA2 = (i * A2_size) + j;
-13         A[pA2] += B[pB3] * C[pC2] * D[pD2];
+14         w[j]  += B[pB3] * C[pC2];
15        }
16      }
17
+18      for (int j = 0; j < n; j++) {
+19        int pD2 = (k * D2_size) + j;
+20        int pA2 = (i * A2_size) + j;
+21        A[pA2] += w[j] * D[pD2];
+22        w[j] = 0.0;
+23      }
24    }
25  }
```

Fig. 9: Source code diff that shows the effect on the generated code of applying the transformation to precompute $B_{ikl}C_{lj}$ into the workspace w.

loop nest. This transformation demonstrates that the workspace transformation can be used to hoist sparse loop-invariant code.

The source code diff in Figure 9 shows the effect of the first workspace transformation on the sparse code that results from compiling the concrete index notation expressions. White background shows unchanged code, red background shows removed code, and green background shows added code. The transformed concrete index notation results in code where the j loop starting on line 9, that multiplies B with D, has been lifted out of the l loop starting on line 7, resulting in fewer multiplications. The cost is that the workspace reduces temporal locality, due to the reuse distance between writing values and reading them back. Our results in Figure 12 shows that this specific transformation results in better performance on two large data sets and reduces performance on a smaller data set. It should therefore be applied judiciously.

The MTTKRP kernel, like the sparse matrix multiplication in Figure 1d, scatters values into the result matrix A. We can observe this from the use of an incrementing assignment statement, $A_{ij} \mathrel{+}= w_j D_{kj}$, and the reason is that the statement is inside one or more forall loops corresponding to a summation. If the matrix A is sparse, then inserts are expensive and the code profits from applying the workspace transformation again to precompute $w_j D_{kj}$ into a workspace v, resulting in the statement

$$\forall_i \left(\forall_j \; A_{ij} = v_j\right) \textbf{ where } \left(\forall_k \left(\forall_j \; v_j \mathrel{+}= w_j D_{kj}\right) \textbf{ where } \left(\forall_{lj} \; w_j \mathrel{+}= B_{ikl} C_{lj}\right)\right)$$

As a result the assignment to A is no longer an incrementing assignment. Instead, the values are scattered into a dense workspace with random access and then copied to the result after a full row of the result has been computed. The source code diff in Figure 10 shows the effect on the compiled code of

```
-1  memset(A, 0, A1_size * A2_size * sizeof(double));
+1  memset(v, 0, v1_size * sizeof(double));
 2  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
 3    int i = B1_crd[pB1];
 4    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
 5      int k = B2_crd[pB2];
 6      memset(w, 0, w1_size * sizeof(double));
 7      for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
 8        int l = B3_crd[pB3];
 9        for (int pC2 = C2_pos[l]; pC2 < C2_pos[l+1]; pC2++) {
10          int j = C2_crd[pC2];
11          w[j] += B[pB3] * C[pC2];
12        }
13      }
14
15      for (int pD2 = D2_pos[k]; pD2 < D2_pos[k+1]; pD2++) {
16        int j = D2_crd[pD2];
-17       int pA2 = (i * A2_size) + j;
-18       A[pA2] += w[j] * D[pD2];
+19       v[j]  += w[j] * D[pD2];
20      }
21    }
22
+23    for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
+24      int j = A2_crd[pA2];
+25      A[pA2] = v[j];
+26      v[j] = 0.0;
+27    }
28  }
```

Fig. 10: Source code diff that shows the effect on the generated code of applying a further transformation to also precompute $w_j D_{kj}$ into the workspace v.

making the result matrix A sparse and precomputing $w_j D_{kj}$ in a workspace v. Both the code from before the transformation (red) and the code after (green) assumes the operand matrices C and D are sparse, as opposed to Figure 9 where C and D were dense. As in the sparse matrix multiplication code, the code after the workspace transformation scatters into a dense workspace v and, when a full row has been computed, appends the workspace nonzeros to the result.

VIII. Evaluation

In this section, we evaluate the effectiveness of the workspace transformation by comparing the performance of sparse kernels with workspaces against hand-written state-of-the-art sparse libraries for linear and tensor algebra.

A. Methodology

All experiments run on a dual-socket 2.5 GHz Intel Xeon E5-2680v3 machine with 12 cores/24 threads and 30 MB of L3 cache per socket, running Ubuntu 14.04.5 LTS. The machine contains 128 GB of memory and runs kernel version 3.13.0 and GCC 5.4.0. For all experiments, we ensure the machine is otherwise idle and report average cold cache performance for single-threaded execution, unless otherwise noted.

We evaluate our approach by comparing performance on linear algebra kernels with Eigen [9] and Intel MKL [8] 2018.0, two high-performance linear algebra libraries. We also compare performance for tensor algebra kernels against the high-performance SPLATT library for sparse tensor factorizations [7]. We obtained real-world matrices and tensors for the experiments in Sections VIII-B and VIII-C from the SuiteSparse Matrix Collection [24] and the FROSTT Tensor Collection [25]. Details of the matrices and tensors used in the experiments are shown

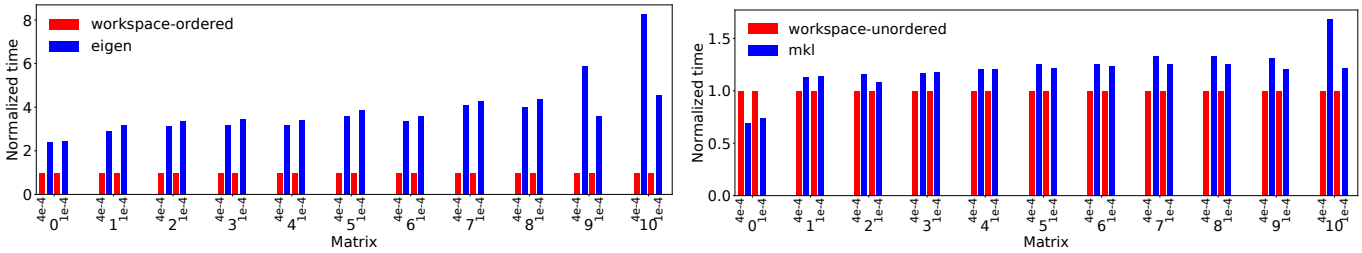


Fig. 11: Sparse matrix multiplication results for the matrices in Table I. Matrix numbers correspond to Table I. We show relative runtime for both sorted (left) and unsorted column entries (right); Eigen's algorithm sorts them while MKL's mkl_sparse_spmm function leaves them unsorted. For the sorted algorithm, we include sorting time in the measurements.

TABLE I: TEST MATRICES FROM THE SUITESPARSE MATRIX COLLECTION [24] AND TENSORS FROM THE FROSTT TENSOR COLLECTION [25].

#	Tensor	Domain	NNZ	Density
0	bcsstk17	Structural	428,650	4E-3
1	pdb1HYS	Protein data base	4,344,765	3E-3
2	rma10	3D CFD	2,329,092	1E-3
3	cant	FEM/Cantilever	4,007,383	1E-3
4	consph	FEM/Spheres	6,010,480	9E-4
5	cop20k	FEM/Accelerator	2,624,331	2E-4
6	shipsec1	FEM	3,568,176	2E-4
7	scircuit	Circuit	958,936	3E-5
8	mac-econ	Economics	1,273,389	9E-5
9	pwtk	Wind tunnel	11,524,432	2E-4
10	webbase-1M	Web connectivity	3,105,536	3E-6
	Facebook	Social Media	737,934	1E-7
	NELL-2	Machine learning	76,879,419	2E-5
	NELL-1	Machine learning	143,599,552	9E-13

in Table I. For synthetic inputs (used, for example, in our sparse matrix addition benchmark), we constructed the synthetic sparse inputs using the random matrix generator in taco, which places nonzeros randomly to reach a target sparsity. All sparse matrices are stored in the compressed sparse row (CSR) format.

B. Sparse Matrix Multiplication

Fast sparse matrix multiplication algorithms use workspaces to store intermediate values [6], [9]. We compare our generated workspace algorithm to the implementations in MKL and Eigen. We compute sparse matrix multiplication with two operands: a real-world matrix from Table I and a synthetic matrix generated with a specific target sparsity and uniform random placement of nonzeros. Eigen implements a *sorted* algorithm, which sorts the column entries within each row so they are ordered, while MKL's mkl_sparse_spmm implements an *unsorted* algorithm—the column entries are unsorted.[1] Because these two algorithms incur different costs, we compare to a workspace variant of each; for the sorted algorithm, we include sorting time. In both cases, the workspace algorithm fuses assembly of the output matrix with the computation. Note that our previous implementation[2] does not generate sparse matrix multiplication, because it does not support insertion into sparse results, so we omit it from this comparison.

[1]According to MKL documentation, its sorted algorithms are deprecated and should not be used.

[2]as of Git revision bf68b6

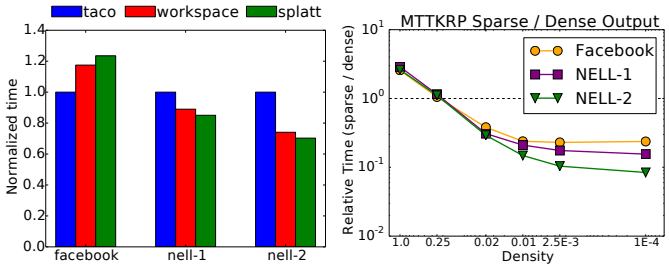
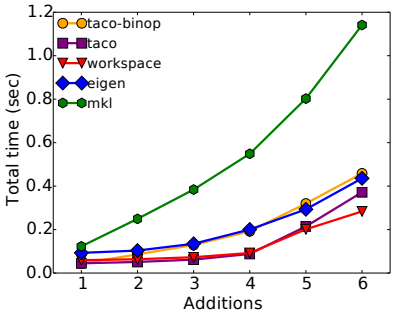


Fig. 12: Left: MTTKRP running times, normalized to taco time, running on in parallel on a single socket. Right: Relative MTTKRP compute time as the density of matrix operands varies, for the three test sparse tensors, for MTTKRP with sparse output and matrix operands, compared with MTTKRP with dense output and matrix operands. The latter comparison uses single-threaded performance, as we have not implemented a parallel MTTKRP with sparse output.

Figure 11 shows running times for sparse matrix multiplication for each matrix in Table I multiplied by a synthetic matrix of nonzero densities 1E-4 and 4E-4, using our workspace implementation. On average, Eigen is slower than our approach, which generates a variant of Gustavson's matrix multiplication algorithm, by 4× and 3.6× respectively for the two sparsity levels. For the unsorted algorithm, we compare against MKL, and find that our performance is 28% and 16% faster on average. The generated workspace algorithm is faster (by up to 68%) than MKL's hand-optimized implementation in all but one case, which is 31% slower.

C. Matricized Tensor Times Khatri-Rao Product

Matricized tensor times Khatri-Rao product (MTTKRP) is used to compute tensor factorizations in data analytics. The three-dimensional version takes as input a sparse 3-tensor and two matrices, and outputs a matrix. Figure 12 shows the results for our workspace algorithm on three input tensors, compared to taco and the hand-coded SPLATT library. We compare parallel single-socket implementations, using `numactl` to restrict execution to a single socket.



Code	Assembly	Compute
taco bin	247	211
taco	190	182
workspace	190	93.3
Eigen		436
MKL		1141

Fig. 13: Left: Scaling plot that shows the time in seconds to assemble and compute n matrix additions with Eigen, MKL, taco binary operations, a single multi-operand taco function, and with workspaces. For n additions, $n+1$ matrix operands are added together. Right: Breakdown of sparse matrix addition time in milliseconds for adding 7 matrices, for all codes. The operands are generated random sparse matrices of densites (fraction of nonzeros) 2.56E-02, 1.68E-03, 2.89E-04, 2.50E-03, 2.92E-03, 2.96E-02, and 1.06E-02.

For the NELL-1 and NELL-2 tensors, the workspace algorithm outperforms the merge-based algorithm in taco by 12% and 35% respectively, and is within 5% of the hand-coded performance of SPLATT. On the smaller Facebook data set, the merge algorithm is faster than both our implementation and SPLATT's. Different inputs perform better with different algorithms, which demonstrates the advantage of being able to generate both versions of the algorithm. Since MTTKRP is the biggest bottleneck of the widely used alternating least squares tensor decomposition algorithm, accounting for 69–99% of the computation time [26], we expect that speeding up MTTKRP using the workspace transformation will provide similar speedups to the overall runtime of tensor decomposition.

D. MTTKRP with Sparse Matrices

It is useful to support MTTKRP where both the tensor and matrix operands are sparse [27]. If the result is also sparse, then MTTKRP can be faster since it only needs to iterate over nonzeros. The code is tricky to write, however, and cannot be generated by the current version of taco. In this section, we evaluate a workspace implementation of sparse MTTKRP. Because we have not implemented a parallel version of MTTKRP with sparse outputs, we perform this comparison with single-threaded implementations of both MTTKRP versions.

Which version is faster depends on the density of the sparse operands. Figure 12 shows experiments that compare the compute times for MTTKRP with sparse matrices against MTTKRP with dense matrices, as we vary the density of the randomly generated input matrices. For each of the tensors, the crossover point is at about 25% nonzero values, showing that such a sparse algorithm can be faster even with only a modest amount of sparsity in the inputs. At the extreme, matrix operands with density 1E-4 can obtain speedups of 4.5–11× for our three test tensors.

E. Sparse Matrix Addition

To demonstrate the utility of workspaces for sparse matrix addition, we show that the algorithm scales as we increase the number of operands. In Figure 13, we compare the workspace algorithm to taco computing one addition at a time (as a library would be implemented), taco generating a single function for the additions, Intel MKL (using its inspector-executor implementation), and Eigen. We pre-generate k matrices with target sparsities chosen uniformly randomly from the range [1E-4, 0.01] and always add in the same order and with the same matrices for each library. Note also that the x-axis shows the number of additions, which is always one more than the number of operands.

The results of this experiment show two things. First, that the libraries are hampered by performing addition two operands at a time, having to construct and compute multiple temporaries, resulting in less performance than is possible using code generation. Even given this approach, taco is faster than Intel MKL by 2.8× on average, while Eigen and taco show competitive performance.

Secondly, the experiment shows the value of being able to produce both merge-based and workspace-based implementations of sparse matrix addition. At up to four additions, the two versions are competitive, with the merge-based code being slightly faster. However, with increasing numbers of additions, the workspace code begins to outperform the taco implementation, showing an increasing gap as more operands are added. Figure 13 (right) breaks down the performance of adding 7 operands, separating out assembly time for the taco-based and workspace implementations. For this experiment, we reuse the matrix assembly code produced by taco to build the output, but compute using a workspace. Most of the time is spent in assembly, which is unsurprising, given that assembly requires memory allocations, while the computation performs only point-wise work without the kinds of reductions found in MTTKRP and sparse matrix multiplication.

IX. Related Work

Related work is divided into work on dense and sparse tensor algebra compilation, work on general loop optimization, alternatives to concrete index notation, and manual workspace transformations in specific matrix and tensor kernels.

There has been much work on optimizing dense matrix and tensor computations [28]–[31]. Researchers have also worked on compilation and code generation of sparse matrix computations, including Bik and Wijshoff [1], the Bernoulli system [2], and SIPR [32]. Furthermore, this paper builds on our recent work on a sparse tensor algebra theory [4], [33] and tool [34] that compiles tensor index notation on sparse and dense tensors, and that we have extended to cover many more sparse tensor formats [5]. These sparse compilation approaches, however, did not generate sparse code with tensor workspaces to improve performance.

One use of the workspace transformation in loop nests, in addition to removing multi-way merge code and scatters into sparse results, is to split apart computation that may

take place at different loop levels. This results in operations being hoisted to a higher loop nest. Loop invariant code motion has a long history in compilers, going back to the first FORTRAN compiler in 1957 [35]. Recently, researchers have found new opportunities for removing redundancy in loops by taking advantage of high-level algebraic knowledge [36]. Our workspace transformation applies to sparse tensor algebra and can remove loop redundancies from sparse code with indirect-access loop bounds and many conditional branches.

The polyhedral model was designed to optimize dense loop nests with affine loop bounds and affine array accesses. Sparse codes, however, have while loops and indirect array accesses. Recent work extended the polyhedral model to these situations [3], [15], [16], [37], [38], using a combination of compile-time and runtime techniques. However, the space of loop nests on hierarchical indirect array accesses is complicated and it is difficult for compilers to determine when optimizations are applicable. For example, the work of Spek and Wijshoff [39] and Venkat et al. [3] transforms sparse code into dense loops and introduces a conditional to ensure only nonzero entries are computed. In this way, traditional dense loop transformations such as tiling can be applied before transforming the code back to operate on sparse structures. However, this technique does not introduce dense tensor temporaries like the workspace transformation but tackles the orthogonal problem of transforming already-existing sparse accesses. Our workspace transformation applies to sparse tensor algebra at the concrete index notation level before sparse code is generated. This makes it possible to perform aggressive optimizing transformations while ensuring the legality of the generated code.

The value of concrete index notation is to specify computation order and temporaries within that order in tensor expressions, while semantically excluding loop data dependencies. Two similar notations from the recent literature are Tensor Comprehensions (TC) [13] and GLORE's LER [36]. The first difference between the concrete index notation and these languages is that operands can be sparse, and sparse expressions are optimized before sparse indices, conditionals, and while loops are introduced. Furthermore, TC consists of a sequence of index notation expressions where loops are implied by index variables, so it cannot express workspaces in inner dimensions (TC's where clauses set index ranges). Our notation expresses ordering the same way as GLORE's LER notation but expresses temporary computations differently. First, our notation uses where clauses to express all temporary computations including reductions, whereas LER has reduction expressions. Second, our notation has scalar assignments inside loop expressions, whereas LER has tensor assignments outside loop expressions. Thus, LER expresses temporary computations as separate statements, and cannot express temporaries inside loop nests without further loop fusion in another notation.

The first use of dense workspaces for sparse matrix computations is Gustavson's sparse matrix multiplication implementation, that we recreate with the workspace transformation to produce the code in Figure 1d [6]. A workspace used for accumulating temporary values is referred to as an expanded real accumulator in [40] and as an abstract sparse accumulator data structure in [41]. Dense workspaces and blocking are used together to produce fast parallel code by Patwary et al. [12]. They also tried a hash map workspace, but report that it did not have good performance for their use. Furthermore, Buluç et al. use blocking and workspaces to develop sparse matrix-vector multiplication algorithms for the CSB data structure that are equally fast for Ax and A^Tx [42]. Im et. al. and Vuduc et. al. store regular dense blocks in the BCSR sparse matrix format and present several heuristics to choose block sizes for sparse matrix operations and storage [43], [44]. Finally, Smith et al. uses a workspace to hoist loop-invariant code in their implementation of MTTKRP in the SPLATT library [7]. We re-create this optimization with the workspace transformation and show the resulting source code in Figure 9.

X. Conclusion

This paper presents a transformation to introduce workspaces into sparse code to remove insertion into sparse results, to remove conditionals, and to hoist loop-invariant computations. The transformation is expressed in a new concrete index notation IR for describing how tensor index notation should execute. The transformation enables a new class of sparse tensor computations with sparse results and improves performance of other tensor computations to match state-of-the-art hand-optimized implementations. We believe the importance of workspaces will increase in the future as combining new tensor formats will require workspaces as glue. Furthermore, we believe concrete index notation can grow into a language for general tensor code transformation, including loop tiling, strip-mining, and splitting. Combined with a full-fledged scheduling language to command these concrete index notation transformations, the resulting system would separate algorithm from schedule. This would allow end users to specify the computation they want, in tensor index notation, while the specification for how it should execute can be specified by performance experts, autotuning systems, machine learning, or heuristics.

Acknowledgment

We thank Samuel Gruetter, Ryan Senanayake, and Stephen Chou for helpful discussion, suggestions, and reviews. This work was supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the Toyota Research Institute; the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121 and a Computational Science Graduate Fellowship DE-FG02-97ER25308; the National Science Foundation under Grant No. CCF-1533753; and DARPA under Award Number HR0011-18-3-0007. 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

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

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

[3] 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*, ser. PLDI 2015, 2015, pp. 521–532.

[4] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017.

[5] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, Oct. 2018.

[6] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, 1978.

[7] 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)*, May 2015, pp. 61–70.

[8] Intel, "Intel math kernel library reference manual," 630813-051US, 2012. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf, Tech. Rep., 2012.

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

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

[11] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, Jul. 2012.

[12] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

[13] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," Feb. 2018.

[14] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3176–3190, Dec. 2014.

[15] M. M. Strout, G. Georg, and C. Olschanowsky, "Set and Relation Manipulation for the Sparse Polyhedral Framework," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Sep. 2012, pp. 61–75.

[16] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati, "FADAlib: an open source C++ library for fuzzy array dataflow analysis," *Procedia Computer Science*, vol. 1, no. 1, pp. 2075–2084, May 2010.

[17] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems," *SIAM review*, vol. 33, no. 3, pp. 420–460, 1991.

[18] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Studies in Applied Mathematics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[19] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[20] A. H. Phan and A. Cichocki, "Tensor decompositions for feature extraction and classification of high dimensional datasets," *Nonlinear theory and its applications, IEICE*, vol. 1, no. 1, pp. 37–68, 2010.

[21] J. Möcks, "Topographic components model for event-related potentials and some biophysical considerations," *IEEE transactions on biomedical engineering*, vol. 35, no. 6, pp. 482–484, 1988.

[22] A. Shashua and A. Levin, "Linear image coding for regression and classification using the tensor-rank principle," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–I.

[23] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[25] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[26] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse cp decomposition for higher-order tensors," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, vol. 00, May 2017, pp. 1048–1057.

[27] S. Smith, A. Beri, and G. Karypis, "Constrained tensor factorization with accelerated ao-admm," in *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 2017, pp. 111–120.

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

[29] M. J. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982, aAI8303027.

[30] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 4, pp. 424–453, 1996.

[31] 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*, vol. 104, no. 2, pp. 211–228, 2006.

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

[33] D. Lugato, F. Kjolstad, S. Chou, S. Amarasinghe, and S. Kamil, "Taco: Compilation et génération de code dexpressions tensorielles," *AVANCÉES*, p. 52, 2018.

[34] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "taco: A tool to generate tensor algebra kernels," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 943–948.

[35] J. Backus, "The history of fortran i, ii, and iii," in *History of programming languages I*. ACM, 1978, pp. 25–74.

[36] Y. Ding and X. Shen, "Glore: Generalized loop redundancy elimination upon ler-notation," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 74:1–74:28, Oct. 2017.

[37] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 41.

[38] M. M. Strout, M. Hall, and C. Olschanowsky, "The sparse polyhedral framework: Composing compiler-generated inspector-executor code," *Proceedings of the IEEE*, no. 99, pp. 1–15, 2018.

[39] H. L. Van Der Spek and H. A. Wijshoff, "Sublimation: expanding data structures to enable data instance specific optimizations," in *Languages and Compilers for Parallel Computing*. Springer, 2011, pp. 106–120.

[40] S. Pissanetzky, *Sparse Matrix Technology-electronic edition*. Academic Press, 1984.

[41] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[42] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "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*. ACM, 2009, pp. 233–244.

[43] E.-J. Im and K. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *Computational Science ICCS 2001*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, May 2001, pp. 127–136.

[44] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply." IEEE, 2002, pp. 26–26.