Sparse Tensor Algebra Optimization with Workspaces

Fredrik Kjolstad
MIT
fred@csail.mit.edu

Peter Ahrens
MIT
pahrens@csail.mit.edu

Shoaib Kamil
Adobe Research
kamil@adobe.com

Saman Amarasinghe
MIT
saman@csail.mit.edu

Abstract

This paper shows how to extend sparse tensor algebra compilers to introduce temporary tensors, called workspaces, to avoid the inefficiencies of accessing sparse data structures. 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, sparse matrix multiplication and MTTKRP with sparse output were not supported by prior tensor algebra compilers and the performance of MTTKRP with dense output improves by up to 35%.

Keywords sparse tensor algebra, optimization, temporaries

1 Introduction

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

Workspaces provide further opportunities for optimizing loops that compute on sparse tensors. A sparse tensor contains mostly zeros and is stored in a compressed data structure. Dense workspaces can drastically reduce access 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 lower dimensionality to keep memory cost down, we can reduce the cost of access, insertion, and replace merge loops with random accesses.

Prior work on sparse tensor compilation describes how to optimize sparse imperative code [4, 18, 35] and how to generate sparse code from high-level tensor index notation [6, 16]. These papers do not, however, consider optimizations that introduce temporary tensors. Such optimizations are important in many sparse tensor kernels, such as tensor additions, sparse matrix multiplication (where all matrices are sparse) [12], and the matricized tensor times Khatri-Rao product (MTTKRP) used to factorize sparse tensors [30]. Without compiler support for workspaces we leave performance on the table. In fact, the sparse matrix multiplication kernel is asymptotically slower without workspaces [12].

This paper presents a compiler transformation that introduces temporary tensor workspaces into sparse code generated from tensor index notation. The workspace transformation is expressed in a intermediate representation (IR) called concrete index notation, that precisely describes when computations occur and where results are stored.

The workspace transformation is programatically invoked through a scheduling API, giving the user control of when to apply it in custom kernels. We do not consider how to automatically determine when the transformation should be applied in this paper. Such a policy system, however, can be cleanly built on top of our scheduling API in future work.

Our main contributions are:

Concrete Index Notation We introduce a new tensor algebra IR that specifies loop order and temporary workspace variables (Section 3).

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 4).

CGO'19 (preprint), Feb. 16–20, 2019, Washington DC, USA
2019.

Compilation We show how to compile concrete index notation with workspaces to sparse code, building on prior work by Kjolstad et al. [16] (Section 5).

Case Studies Throughout the paper 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 MKL [14], Eigen [11], and SPLATT [30] high-performance libraries, including speedups of 4× over Eigen and 1.28× over MKL for sparse matrix multiplication.

2 Motivating Example

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

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

Sparse kernel code depends on the operand 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 [11, 14, 19]. In the CSR format, each matrix row is compressed (only nonzeros are stored). This requires two index arrays to describe the matrix 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_idx` and a value array `B`. The array `B_idx` contains the column coordinates of nonzero values in corresponding positions in `B`. The array `B_pos` stores the position of the first column coordinate of each row in `B_idx` and a sentinel with the number of nonzeros in the matrix (nnz). Thus, contiguous values in `B_pos` store the beginning and end [inclusive-exclusive) of a row in the arrays `B_idx` and `B`. For example, the column coordinates of the third row are stored in `B_idx` 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 algorithms compute the matrix multiply using a *linear combination of rows* matrix multiply, that compute rows of A as sums of rows from C scaled by rows 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 [12]. Second, linear combinations of rows work 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.

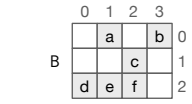
The matrix multiplication kernel in Figure 1c has two sparse CSR operand matrices but a dense result. Because it contains the sub-expression B_{ik} it iterates over B's sparse matrix data structure with the loops over i (line 1) and k (lines 2–3). The loop over i is dense because the CSR format stores 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_idx`. We have highlighted B's index arrays in Figure 1c.

The kernel in Figure 1d also has a sparse CSR result that saves memory when many of the values are zeros. The sparse result matrix complicated the kernel because the assignment to A (line 6) 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. Such workspaces and the accompanying loop transformations are the subject of this paper.

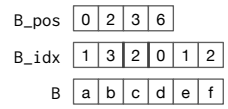
A workspace is a dense temporary tensor, with fast insertion and random access, and lower dimensionality than the result to keep down memory usage. Because values can be scattered efficiently into a dense workspace, the loop nest k, j (lines 2–8) in Figure 1d looks similar to the kernel in Figure 1c. 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 10–14). 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 increases performance when assembly can be moved out of inner loops, common in simulation codes, but it is also possible to simultaneously assemble and compute.

By extending the open-source taco system [16] with one new method, we can add workspace into kernels. We apply the `workspace` method to the multiplication and provide as arguments the index variable to apply the workspace optimization to and the format of the workspace. The following C++ code would generate the code in Figure 1d:

```
1 Format CSR({dense, sparse});
2 TensorVar A(CSR), B(CSR), C(CSR);
3 IndexVar i, k, j;
4
5 IndexExpr mul = B(i,k) * C(k,j);
6 A(i,j) = sum(k, mul);
```



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



(b) Sparse CSR index of B

```
for (int i = 0; i < m; i++) {
  for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
    int k = B_idx[pB];
    for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
      int j = C_idx[pC];
      A[i*o+j] += B[pB] * C[pC];
    }
  }
}
```

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

```
1  for (int i = 0; i < m; i++) {
2    for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
3      int k = B_idx[pB];
4      for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
5        int j = C_idx[pC];
6        workspace[j] += B[pB] * C[pC];
7      }
8    }
9
10   for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {
11     int j = A_idx[pA];
12     A[pA] = workspace[j];
13     workspace[j] = 0.0;
14   }
15 }
```

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

Figure 1. Figure (a) shows a sparse matrix and (b) its compressed data structure. Futhermore, figure (c) shows a sparse matrix multiplication with a dense result and (d) with a sparse result. Sparse matrices do not support $O(1)$ insert, so we introduce a dense temporary workspace. Code to zero A and the workspace have been omitted, and result indices have been pre-assembled.

```
7
8 mul.workspace({j}, Format(dense));
```

Lines 1–3 creates a CSR format, three tensor variables, and three index variables. Lines 5–6 define a sparse matrix multiplication and line 8 declares that the multiplication should be pre-computed in a workspace across the j loop.

We expose the workspace transformation as an API in the spirit of Halide [26]. To separate policy (what to do) from mechanism (how to do it) we define the `workspace` method, that users must choose how to use. We make no attempt in this paper to automate the process of deciding when to apply the transformation. It is, however, interesting future work to build an automated policy system on top of our API, with techniques such as heuristics, models, autotuning, or ML.

3 Concrete Index Notation

Index notation is a popular input language for tensor algebra code generators and frameworks [16, 31, 34]. It describes what tensor operations should do independently of how they should be computed or how tensors should be stored. Thus, optimization decisions are not mixed with algorithmic descriptions. This is possible partly because index notation restricts the space of operations that can be described.

While index notation is a suitable input language for a tensor operations, it is unsuitable as a compiler IR. The reason is that it does not specify the order of execution or temporaries 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 [3, 32], or iteration graphs [16]. These representations, however, are too general to conveniently apply the workspace optimizations 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 and where to use temporaries and their formats, without details of how to process the sparse storage formats of operands. In the compiler software stack, concrete index notation is an intermediate representation between index notation and low level imperative IR, as shown in Figure 4. A benefit of this design is that we can transform concrete index notation instead of sparse imperative code, which is only generated when concrete index notation is lowered.

Concrete index notation has four main statement types. The assignment statement assigns an expression result to a tensor element, 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 temporary storage.

We return to the running example of linear combinations of rows matrix multiply. We express this algorithm in concrete index notation by operating in i, k, j order. The concrete index notation is written on the left, and the 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
```

The **assignment** statement $A_{ij} \mathrel{+}= B_{ik}C_{kj}$ modifies the value of a single 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. The **forall** statements $\forall_i\forall_k\forall_j$, abbreviated as $\forall_{ikj}$, specify iteration order. Finally, result tensors are implicitly initialized to zero.

This algorithm repeatedly computes and adds scaled rows to the matrix A. If A is sparse, however, it is very expensive to repeatedly add rows. We can use the **where** statement of concrete index notation to add 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 introduces a dense temporary vector w to hold the partial sums of rows on the right-hand producer side, that is used on the left-hand consumer side.

Suppose now that we want 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 reuse:

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

```
for i ∈ I
  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, tensors defined on the left of a sequence can be modified on the right of the sequence statement.

4 Workspace Transformation

The workspace transformation pre-computes tensor algebra sub-expressions 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 Merge code to simultaneously iterate over sparse tensors contains expensive conditionals and loops. By pre-computing sub-expressions into dense workspaces, the merge code is simplified (see Figure 2).

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 (see Figure 1).

Hoist loop invariant code Computing everything in the inner-most loop can result in redundant computations. Pre-computing a sub-expression in a separate loop and storing the results in a workspace can hoist parts of an inner loop (see Figure 6b).

The transformation rewrites concrete index notation to pre-compute a sub-expression. The target sub-expression is computed and stored in a temporary tensor (the workspace). Next, the main expression (where the sub-expression has been replaced by an access to the workspace). Surrounding forall statements are relocated to compute either the sub-expression or the main expression (whichever is valid) until we reach a forall statement which cannot be moved. The effect is that the original statement is split in two; one statement produces values for the other through the workspace.

Let (S, E, I) be the inputs to the workspace transformation, where S is a statement that does not contain sequences, I is a set of index variables, and E is an expression contained

```
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_idx[pB2];
    int jC = C2_idx[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, n*sizeof(double));

  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_idx[pB2];
    w[j] = B[pB2];
  }

  for (int pC2 = C2_pos[i]; pC2 < C2_pos[i+1]; pC2++) {
    int j = C2_idx[pC2];
    a[i] += w[j] * C[pC2];
  }
}
```

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

Figure 2. 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 optimization introduces a **where** statement that replaces the merge loop in (a) with two for loops in (b).

in an assignment statement S_A contained in S. If S_A is an incrementing assignment statement, let $\oplus$ be the associated operator. The workspace transformation then rewrites the statement S_A to precompute E in a workspace as follows:

Let S'_A be S_A where E has been replaced by the access expression w_I where w is a fresh tensor variable.
In S, replace S_A with S'_A **where**$(w_I \oplus= E)$.
Let S_w be this where statement.
while S_w is contained in some forall statement $\forall_j S_w$ **do**
  if j is used in both sides of S_w and $j \in I$ **then**
    Move $\forall_j$ into both sides of S_w.
  else if j is used only in the left side of S_w **then**
    Move $\forall_j$ into the left side of S_w.
  else if j is used only in the right side of S_w **then**
    Move $\forall_j$ into the right side of S_w.
  else
    Stop.
  end if
end while

The transformation can only be applied if every operator that contains E on the right hand side of S_A distributes over $\oplus$. The dimensionality of the resulting workspace is the number

of index variables in I and the dimension sizes are equal to the ranges of those index variables in the existing expression.

Figure 2 shows the effect of applying the workspace optimization on a kernel that computes the inner product of each pair of rows from two matrices, before and after the workspace optimization is applied to the matrix B over j. In this example the optimization causes the while loop over j, which simultaneously iterates over the two rows, to be replaced with a for loop that independently iterates over each of the rows. 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 5.

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 that 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:

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

That is, b is assigned to a followed by an incrementing assignment that adds in c.

The workspace optimization can reuse the result as a workspace if two preconditions are satisfied. The first precondition requires that the forall statements on the two sides of the resulting sequence statement be the same, that is, the optimization does not hoist computation out of a loop. This precondition ensures that the result does not get ove-written by its use as a workspace. For example, this precondition is not satisfied by the second workspace optimization to the MTTKRP kernel in Section 6. The second precondition is that the expression E is nested inside at most one operator in S_A, which ensures we can rewrite S_A to an incrementing assignment.

Figure 3 shows a sparse matrix addition with CSR matrices before and after applying the workspace optimization 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 optimization applies to the sub-expression $B_{ij} + C_{ij}$ over j resulting in

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

The second transformation applies to the B_{ij} sub-expression on the right-hand side of the **where**. Without result reuse the result would be

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

```
1  int pA = 0;
2  for (int i = 0; i < m; i++) {
3    int pB2 = B2_pos[i];
4    int pC2 = C2_pos[i];
5    while (pB2 < B2_pos[i+1] && pC2 < C2_pos[i+1]) {
6      int jB = B2_idx[pB2];
7      int jC = C2_idx[pC2];
8      int j = min(jB, jC);
9      if (jB == j && jC == j) {
10       A[pA++] = B[pB2] + C[pC2];
11     }
12     else if (jB == j) {
13       A[pA++] = B[pB2];
14     }
15     else {
16       A[pA++] = C[pC2];
17     }
18     if (jB == j) pB2++;
19     if (jC == j) pC2++;
20   }
21   while (pB2 < B2_pos[i+1]) {
22     A[pA++] = B[pB2++];
23   }
24   while (pC2 < C2_pos[i+1]) {
25     A[pA++] = C[pC2++];
26   }
27 }
```

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

```
1  for (int i = 0; i < m; i++) {
2    for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
3      int i = B2_idx[pB2];
4      w[i] = B[pB2];
5    }
6
7    for (int pC2 = C2_pos[i]; pC2 < C2_pos[i+1]; pC2++) {
8      int i = C2_idx[pC2];
9      w[i] += C[pC2];
10   }
11
12   for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
13    int i = A2_idx[pA2];
14    A[pA2] = w[i];
15    w[i] = 0.0;
16   }
17 }
```

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

Figure 3. 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 $B_{ij} + C_{ij}$, and the second application is to B and reuses the workspace with a sequence statement. These optimizations replace the inner merge loop in (a) with two loops that accumulate the operands into w and one to copy values from w to A in (b).

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

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

5 Compilation

This paper proposes concrete index notation and workspace transformations as extensions to the tensor algebra compiler system of Kjolstad et al. [16], This section the transformations to and from concrete index notation, called **concretization** and **code generation**.

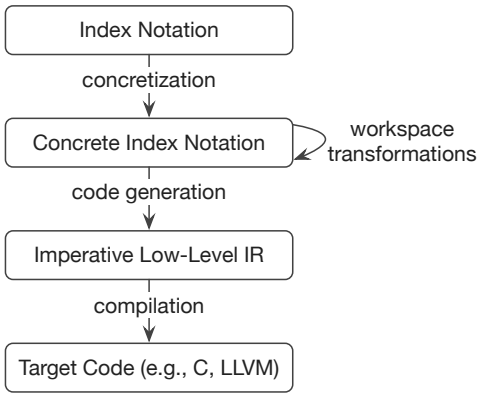


Figure 4. Compiler stages from index notation, through concrete index notation with workspace transformations, to low-level imperative IR and target code.

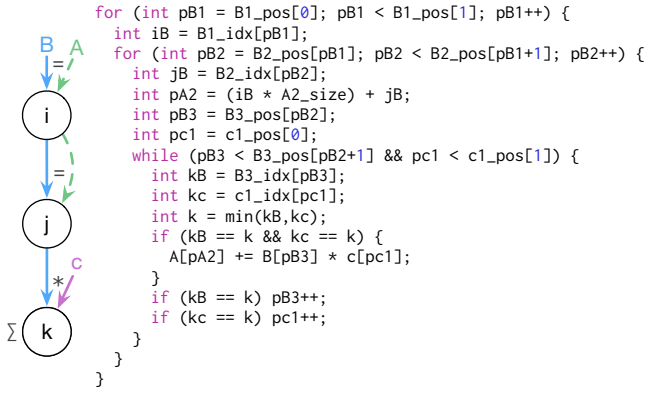


Figure 5. Iteration graph and sparse code of a tensor-vector multiplication $\forall_{ijk}\ A_{ij} \mathrel{+}= B_{ijk}c_k$, where B and c are sparse.

Figure 4 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 2, these transformations are programmatically asked for by the end user or by a future policy system.

The first IR transformation in the compilation workflow concretizes index notation in two steps:

Insert forall statements for each index variable in the index notation expression. This step ensures the nesting of the forall statements is such that sparse tensor formats are traversed in order.

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

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

The second IR transformation lowers concrete index notation to sparse imperative C-like code. This algorithm uses the iteration graph and merge lattice concepts introduced in Kjolstad et al. [16], but is simpler than the algorithm described there. The reason for the simplicity is that it lowers concrete index notation statements and therefore does not have to deduce at each recursive step what sub-expressions that are available to emit.

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

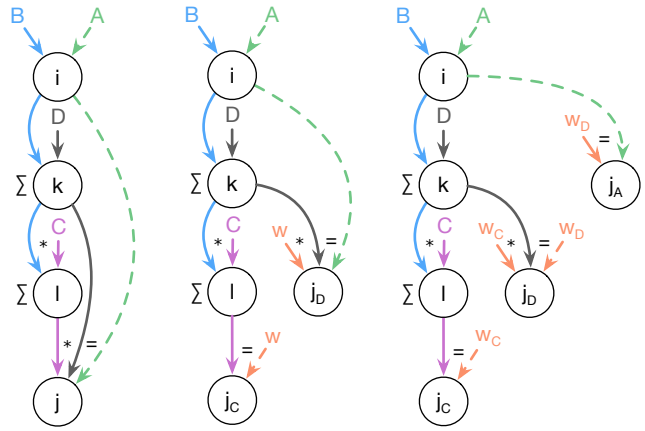
The complexity of sparse code generation is isolated to the code generation of forall statements, as these must simultaneously coiterate over data structures. To understand these complicated iteration patterns we will use iteration graphs [16]. Figure 5 shows the iteration graph of a tensor-vector multiplication $\forall_{ijk}\ A_{ij} \mathrel{+}= B_{ijk}c_k$ (left), and generated sparse C code when the tensor B and vector c are sparse while the result matrix A is dense. The iteration graph shows the hierarchical sparse data structures of the tensors imposing dependencies on the index variables used to access them. A tensor path through index variables indicate those index variables are used to access the tensor. Specifically, this means that loops generated from an index variable must traverse the hierarchical data structures whose paths go through the index variable. For example, the loop generated for j must only traverse the data structure of B (a is a result), while the loop generated for k must simultaneously traverse the third hierarchy level of the B data structure and c.

Forall statements are lowered using merge lattices to generate merge code the same way as in [16]. We refer to their 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 sub-statements, 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 sub-statement is rewritten to remove them by symbolically setting them to zero. Thus, we only recurse on the parts of the statement that are not exhausted, which simplifies the algorithm.

Finally, the code to assemble index data structures requires the same loops as the code to compute components. Inside these loops, however, the compiler must also emit statements to record coordinates in `idx` arrays and to record the number of coordinates per tensor slice in `pos` arrays.

6 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 algorithm for computing the canonical polyadic decomposition



(a) Original **(b)** w_j += $B_{ikl}C_{lj}$ **(c)** v_j += w_jD_{kj}

Figure 6. The iteration graphs of the MTTKRP expression ($A_{ij} = \sum_{kl} B_{ikl}C_{lj}D_{kj}$) before transformation (original), after the first transformation to workspace $B_{ikl}C_{lj}$, and after the second transformation to also workspace w_jD_{kj}. The first transformation removes the edge between j_D and l, which eliminates loop-invariant work. The second transformation is useful when A is sparse, to replace insert into it with inserts to a workspace with faster random insert.

```
 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_idx[pB1];
 4   for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
 5     int k = B2_idx[pB2];
 6     for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
 7       int l = B3_idx[pB3];
 8       for (int jC = 0; jC < C2_size; jC++) {
 9         int pC2 = (l * C2_size) + jC;
-10         int pD2 = (k * D2_size) + jC;
-11         int pA2 = (i * A2_size) + jC;
-12         A[pA2] += B[pB3] * C[pC2] * D[pD2];
+13         w[jC]  += B[pB3] * C[pC2];
14       }
15     }
16
+17     for (int jD = 0; jD < D2_size; jD++) {
+18       int pD2 = (k * D2_size) + jD;
+19       int pA2 = (i * A2_size) + jD;
+20       A[pA2] += w[jD] * D[pD2];
+21       w[jD] = 0.0;
+22     }
23   }
24 }
```

Figure 7. Source code diff that shows the effect on the compiled code of applying the transformation to pre-compute $B_{ikl}C_{lj}$ into the workspace w.

of tensors [13]. This decomposition generalizes the singular value decomposition to higher-order tensors and has applications in data analytics [7], machine learning [23], neuroscience [21], image classification and compression [27], and other fields [17]. In this section we apply the workspace transformation twice to optimize MTTKRP. The first application results in a kernel roughly equivalent to the hand-optimized MTTKRP implementation in SPLATT [30], while the second application enables MTTKRP with sparse matrices.

The MTTKRP kernel for factorizing 3-order tensors is expressed in tensor index notation as $A_{ij} = \sum_{kl} B_{ikl}C_{lj}D_{kj}$. We multiply 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 expression for MTTKRP is

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

and Figure 6a shows the corresponding iteration graph. When we apply the workspace transformation to $B_{ikl}C_{lj}$ at j, the expression is transformed to pre-compute this sub-expression in the workspace w:

$$\forall_{ik}\ \left(\forall_j\ A_{ij}\ \text{+=}\ w_jD_{kj}\right)\ \textbf{where}\ \left(\forall_{lj}\ w_j\ \text{+=}\ B_{ikl}C_{lj}\right).$$

Figure 6b shows the corresponding iteration graph. The workspace transformation has split the j index variable in two, j_C and j_D, corresponding to the forall statements on either side of the where statement. Notice that there are no edges between l and j_D. This was our purpose when applying the transformation, as it means j_D will be computed at a higher level in the resulting loop nest. This demonstrates that the workspace transformation can be used to hoist sparse loop-invariant code.

The source code diff in Figure 7 shows the effect of the first workspace transformation on the sparse code that result 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_C loop, that multiplies B with D, has been lifted out of the l loop, resulting in fewer multiplication. The cost is that the workspace reduces temporal locality, due to the reuse distance between writing values to it and reading them back. Our results in Figure 10 shows that this specific optimization 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 see this in the iteration graphs in Figure 6b by observing that the index variable j_D on the result path (green dashed) have a reduction variable ancestor (k). If the matrix A is sparse, then inserts are expensive and the code profits from applying the workspace transformation again to pre-compute w_jD_{kj} in a workspace v:

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

```
- 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_idx[pB1];
  4    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
  5      int k = B2_idx[pB2];
  6      for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
  7        int l = B3_idx[pB3];
  8        for (int pC2 = C2_pos[l]; pC2 < C2_pos[l+1]; pC2++) {
  9          int jC = C2_idx[pC2];
 10          w[jC] += B[pB3] * C[pC2];
 11        }
 12      }
 13
 14      for (int pD2 = D2_pos[k]; pD2 < D2_pos[k+1]; pD2++) {
 15        int jD = D2_idx[pD2];
- 16        int pA2 = (i * A2_size) + jD;
- 17        A[pA2] += w[jD] * D[pD2];
+ 18        v[jD]  += w[jD] * D[pD2];
 19      }
 20      memset(w, 0, C2_size*sizeof(double));
 21    }
 22
+ 23    for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
+ 24      int jA = A2_idx[pA2];
+ 25      A[pA2] = v[jA];
+ 26      v[jA] = 0.0;
+ 27    }
 28  }
```

Figure 8. Source code diff that shows the effect on the compiled code of applying a further transformation to also pre-compute $w_j D_{kj}$ into the workspace v.

Figure 6c shows the resulting iteration graph, and the result index variable j_A no longer has any reduction variable ancestors. The effect is that values are scattered into a dense workspace with random access and copied to the result after a full row of the result has been computed. The source code diff in Figure 8 shows the effect on the compiled code of making the result matrix A sparse and pre-computing $w_j D_{kj}$ 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 7 where C and D were dense. As in the sparse matrix multiplication code, the code after the workspace optimization scatters into a dense workspace v and, when a full row has been computed, appends the workspace nonzeros to the result.

7 Evaluation

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

7.1 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.

Table 1. Test matrices from the SuiteSparse Matrix Collection [8] and tensors from the FROSTT Tensor Collection [29].

#	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

We evaluate our approach by comparing performance on linear algebra kernels with Eigen [11] and Intel MKL [14] 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 [30]. We obtained real-world matrices and tensors for the experiments in Sections 7.2 and 7.3 from the SuiteSparse Matrix Collection [8] and the FROSTT Tensor Collection [29]. Details of the matrices and tensors used in the experiments are shown in Table 1. 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.

7.2 Sparse Matrix Multiplication

Fast sparse matrix multiplication algorithms use workspaces to store intermediate values [11, 12]. 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 1 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. In both cases, the workspace algorithm fuses assembly of the output matrix with the computation. Note that Kjolstad et al.'s implementation[2] does not generate sparse matrix multiplication, because

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

[2] as of Git revision bf68b6

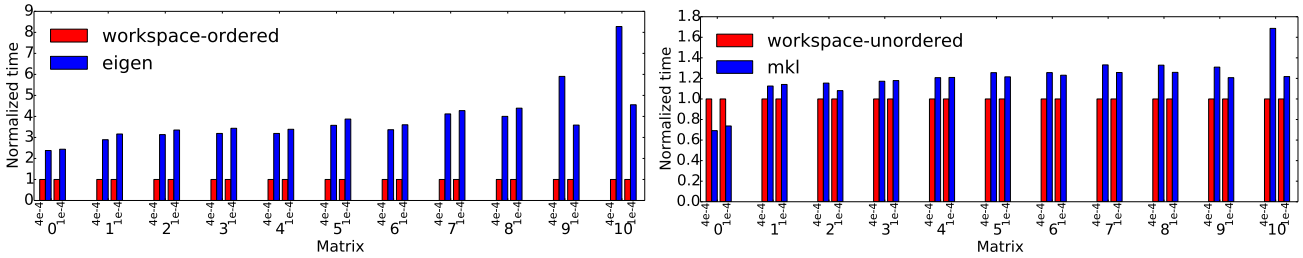


Figure 9. Sparse matrix multiplication results for the matrices in Table 1. Matrix numbers correspond to Table 1. We show relative runtime for both sorted (left) and unsorted column entries (right); Eigen's algorithm sorts them while MKL's `mkl_sparse_spmm` function leaves them unsorted.

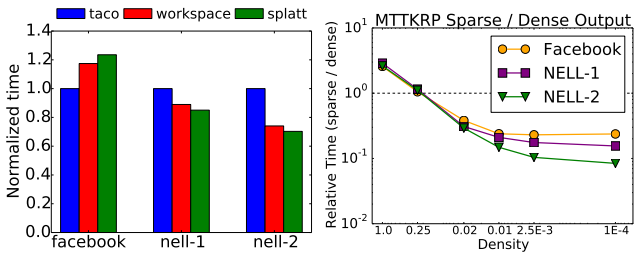


Figure 10. Left: MTTKRP running times, normalized to `taco` time, running on in parallel on a single socket. Right: MTTKRP compute time as the density of matrix operands varies, for the three test sparse tensors. MTTKRP computed with a workspace when the The latter comparison uses single-threaded performance, as we have not implemented a parallel MTTKRP with sparse output.

it does not support insertion into sparse results, so we omit it from this comparison.

Figure 9 shows running times for sparse matrix multiplication for each matrix in Table 1 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.

7.3 Matricized Tensor Times Khatri-Rao Product

Matricized tensor times Khatri-Rao product (MTTKRP) is used to compute generalizations of SVD factorization for tensors in data analytics. The three-dimensional version takes as input a sparse 3-tensor and two matrices, and outputs a matrix. Figure 10 shows the results for our workspace algorithm on three input tensors, compared to `taco` and the hand-coded SPLATT library. We show only compute times, as the assembly times are negligible because the outputs are dense. We compare parallel single-socket implementations, using `numactl` to restrict execution to a single socket.

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 dataset, 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.

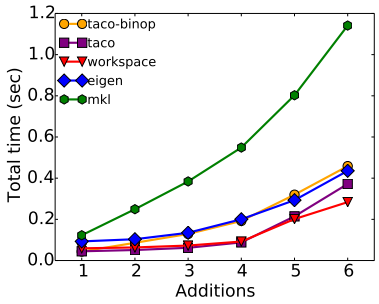
7.4 MTTKRP with Sparse Matrices

It is useful to support MTTKRP where both the tensor and matrix operands are sparse [28]. If the result is also sparse, then MTTKRP can be much 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 use a workspace implementation of sparse MTTKRP. As far as we are aware, ours is the first implementation of an MTTKRP algorithm where all operands are sparse and the output is a sparse matrix. 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 10 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.

7.5 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 11, we compare the workspace algorithm to `taco` computing one addition at a time (as a



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

Figure 11. Left: Scaling plot that shows the time to assemble and compute n matrix additions with Eigen, MKL, taco binary operations, a single multi-operand taco function, and workspaces. 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 2.56E-02, 1.68E-03, 2.89E-04, 2.50E-03, 2.92E-03, 2.96E-02, and 1.06E-02.

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.

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 11 (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.

8 Related Work

There has been much work on optimizing dense matrix and tensor computations [1, 15, 20, 37]. Researchers have also worked on compilation and code generation of sparse matrix computations, including Bik and Wijshoff [4], the Bernoulli system [18], and SIPR [25]. Recently, Kjolstad et al. [16] proposed a tensor algebra compilation theory that compiles tensor index notation on dense and sparse tensors. These sparse compilation approaches, however, did not generate sparse code with tensor workspaces to improve performance.

One use of the workspace optimization 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 [2]. Recently, researchers have found new opportunities for removing redundancy in loops by taking advantage of high-level algebraic knowledge [9]. Our workspace optimization 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 originally designed to optimize dense loop nests with affine loop bounds and affine accesses into dense arrays. Sparse code, however, involves nested indirect array accesses. Recent work extended the polyhedral model to these situations [3, 32, 33, 35, 36], using a combination of compile-time and runtime techniques, but the space of loop nests on nested indirect array accesses is complicated, and it difficult for compilers to determine when optimizations are applicable. For example, the work of Venkat et al. [35] 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 one that operates on sparse structures. However, this technique does not introduce temporary dense tensors like the workspace optimization, and tackles the orthogonal problem of transforming already-existing sparse accesses. Our workspace optimization applies to sparse tensor algebra at the concrete index notation level, before sparse code is generated, making it possible to perform aggressive optimizations while easily ensuring legality.

The first use of dense workspaces for sparse matrix computations is Gustavson's sparse matrix multiplication implementation, that we recreate with the workspace optimization to produce the code in Figure 1d [12]. A workspace used for accumulating temporary values is referred to as an expanded real accumulator in [24] and as an abstract sparse accumulator data structure in [10]. Dense workspaces and blocking are used to produce fast parallel code by Patwary et al. [22]. 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 [5]. Finally, Smith et al.

uses a workspace to hoist loop-invariant code in their implementation of MTTKRP in the SPLATT library [30]. We re-create this optimization with the workspace optimization and show the resulting source code in Figure 7.

9 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 the concrete index notation language can grow into a language for general tensor optimization, including loop tiling, strip-mining, and splitting. Combined with a scheduling language to command these concrete index notation transformations, the resulting system separates algorithm from schedule. This lets end users 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.

Acknowledgments

The authors thanks Stephen Chou and Ryan Senanayake for helpful reviews and suggestions. This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

References

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

[2] John Backus. 1978. The history of FORTRAN I, II, and III. In *History of programming languages I*. ACM, 25–74.

[3] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati. 2010. FADAlib: an open source C++ library for fuzzy array dataflow analysis. *Procedia Computer Science* 1, 1 (May 2010), 2075–2084. https://doi.org/10.1016/j.procs.2010.04.232

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

[5] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 233–244.

[6] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), to appear.

[7] Andrzej Cichocki. 2014. Era of big data processing: A new approach via tensor networks and tensor decompositions. *arXiv preprint arXiv:1403.2048* (2014).

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

[9] Yufei Ding and Xipeng Shen. 2017. GLORE: Generalized Loop Redundancy Elimination Upon LER-notation. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 74 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133898

[10] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.

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

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

[13] Frank L Hitchcock. 1927. The expression of a tensor or a polyadic as a sum of products. *Studies in Applied Mathematics* 6, 1-4 (1927), 164–189.

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

[15] Kenneth E. Iverson. 1962. *A Programming Language*. Wiley.

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

[17] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

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

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

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

[21] Joachim Möcks. 1988. Topographic components model for event-related potentials and some biophysical considerations. *IEEE transactions on biomedical engineering* 35, 6 (1988), 482–484.

[22] Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*. Springer, 48–57.

[23] Anh Huy Phan and Andrzej Cichocki. 2010. Tensor decompositions for feature extraction and classification of high dimensional datasets. *Nonlinear theory and its applications, IEICE* 1, 1 (2010), 37–68.

[24] Sergio Pissanetzky. 1984. *Sparse Matrix Technology-electronic edition*. Academic Press.

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

[26] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. (2012).

[27] Amnon Shashua and Anat Levin. 2001. 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, I–I.

[28] Shaden Smith, Alec Beri, and George Karypis. 2017. Constrained Tensor Factorization with Accelerated AO-ADMM. In *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 111–120.

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

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

[31] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (Dec. 2014), 3176–3190. https://doi.org/10.1016/j.jpdc.2014.06.002

[32] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. 2012. Set and Relation Manipulation for the Sparse Polyhedral Framework. In *Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 61–75. https://doi.org/10.1007/978-3-642-37658-0_5

[33] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 99 (2018), 1–15.

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

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

[36] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 41.

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