Automatic Generation of Sparse Tensor Kernels with Workspaces

Fredrik Kjolstad
MIT
fred@csail.mit.edu

Shoaib Kamil
Adobe Research, USA
kamil@adobe.com

Saman Amarasinghe
MIT
saman@csail.mit.edu

Abstract

Recent advances in compiler theory describe how to compile sparse tensor algebra. Prior work, however, does not describe how to generate efficient code that takes advantage of temporary workspaces. These are often used to hand-optimize important kernels such as sparse matrix multiplication and the matricized tensor times Khatri-Rao product. Without this capability, compilers and code generators cannot automatically generate efficient kernels for many important tensor algebra expressions. We describe a compiler optimization called operator splitting that breaks up tensor sub-computations by introducing workspaces. Our case studies demonstrate that operator splitting is surprisingly general, and our results show that it increases the performance of important generated tensor kernels to match hand-optimized code.

Keywords sparse tensors, tensor algebra, linear algebra, optimization, operator split, workspaces, performance

1 Introduction

Tensor algebra is an important tool for computing on multimode data in domains such as machine learning [1], data analytics [2, 5], engineering [29], and science [14, 15]. Tensors generalize matrices to any number of dimensions and can model both linear and multilinear relationships. Interesting tensors are often sparse, which means most components are zero. Compressing these tensors is often necessary. For example, a tensor encoding Amazon reviews [42] used to predict user response to a new product [31, 33] contains 15 quintillion components where only 1.7 billion are nonzeros.

We recently proposed a compiler theory that shows us how to automatically generate kernels for tensor algebra expressions with both dense and sparse tensors [25]. We implemented the theory in a library and showed that the performance of many generated kernels is competitive with the performance of hand-optimized kernels in existing libraries.

Our previous approach, however, did not generate kernels that take advantage of dense temporary workspace arrays to accumulate temporary values. Such arrays are a common optimization technique in sparse linear algebra algorithms, such as Gustavson's sparse matrix multiplication (SpMM) [16, 18, 38], and was applied to the matricized tensor times Khatri-Rao product (MTTKRP) by Smith et al. [41, 43]. Without workspaces, kernels underperform due to expensive insertions into sparse tensors, branches from code to merge sparse tensors, and redundant loop-invariant work.

This paper introduces a novel compiler optimization called operator splitting on our iteration graph tensor algebra intermediate representation [25]. An operator split breaks up a tensor sub-expression with respect to an index variable and introduces a dense workspace to store intermediate results. Operator splitting reveals that previous proposed algorithms that use workspaces for different purposes are the results of the same transformation. Our case studies show that operator splitting is a general optimization that applies to many important tensor expressions, including sparse matrix multiplication with the linear combination algorithm, sparse matrix addition, and the matricized tensor times Khatri-Rao product.

The key contributions are:

Formulation We identify the lack of workspaces in the tensor compilation literature.

Operator Split We introduce a compiler optimization that removes expensive inserts into sparse results, eliminates merge code, and hoists loop invariant code.

Applicability We define preconditions for applying operator splits and discuss their trade-offs.

Case Studies We show that operator splits recreate several important algorithms with workspaces from the literature and generalizes to important new kernels.

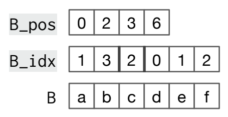
Finally, we evaluate operator splitting by showing that the performance of the resulting code is competitive with hand-optimized implementations with workspaces in the MKL, Eigen, and SPLATT high-performance libraries [17, 22, 43].

2 Motivating Example

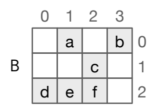
We will use sparse matrix multiplication to introduce sparse tensor data structures, sparse kernels, and the need for workspaces. The ideas, however, generalize to higher-order tensor kernels. Matrix multiplication in linear algebra notation is $A = BC$ and in tensor index notation it is

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

The code in a matrix multiplication kernel depends on the storage formats of the operands and the result. Many matrix storage formats have been proposed in the literature. They can be classified as dense formats that store every matrix



(a) Sparse CSR index of B



(b) Dense $m \times o$ matrix 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        A[i*o+j] += B[pB] * C[pC];
7      }
8    }
9  }
10
11
12
13
14
15
```

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

```c
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];
      workspace[j] += B[pB] * C[pC];
    }
  }

  for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {
    int j = A_idx[pA];
    A[pA] = workspace[j];
    workspace[j] = 0.0;
  }
}
```

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

Figure 1. (a) The CSR sparse matrix index structure of a matrix B, (b) the matrix B, and (c-d) two sparse matrix multiplication kernels written in C. The first kernel (c) operators on matrices stored in the CSR format and its result is dense. The second kernel (d) also operates on CSR matrices, but its result is also a CSR matrix. The second kernel uses a workspace. The code to zero A is omitted and result indices have been pre-assembled. Section 6 discusses the code that assembles result indices.

component and sparse formats that store only the components that are nonzero. Figure 1 shows two linear combination of rows matrix multiplication kernels. We will study this algorithm, instead of the inner product algorithm, because its sparse variant has better asymptotic complexity [18] and because the inputs are conveniently all row major.

Sparse kernels are more involved than dense kernels because they iterate over sparse data structures. Figure 1c shows a sparse matrix multiplication kernel where the result matrix is stored dense row-major and the operand matrices are stored with the compressed sparse row format (CSR) [44].

The CSR format and its column-major CSC sibling are ubiquitous in sparse linear algebra libraries [17, 22, 32] due to their generality and performance. In the CSR format, each matrix row is compressed (only nonzero components are stored). This requires two index arrays to describe the matrix coordinates and positions of the nonzeros. Figure 1a shows the CSR data structure of matrix B in Figure 1b. It consists of the index arrays `B_pos` and `B_idx` and a value array `B`. The array `B_idx` contains the column coordinate of each nonzero value in the corresponding position in `B`. The array `B_pos` stores the position of the first column coordinate of each row in `B_idx`, as well as a sentinel with 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_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]`). Moreover, some libraries require the entries within each row to be sorted in order of ascending coordinate value, which results in faster performance for some algorithms.

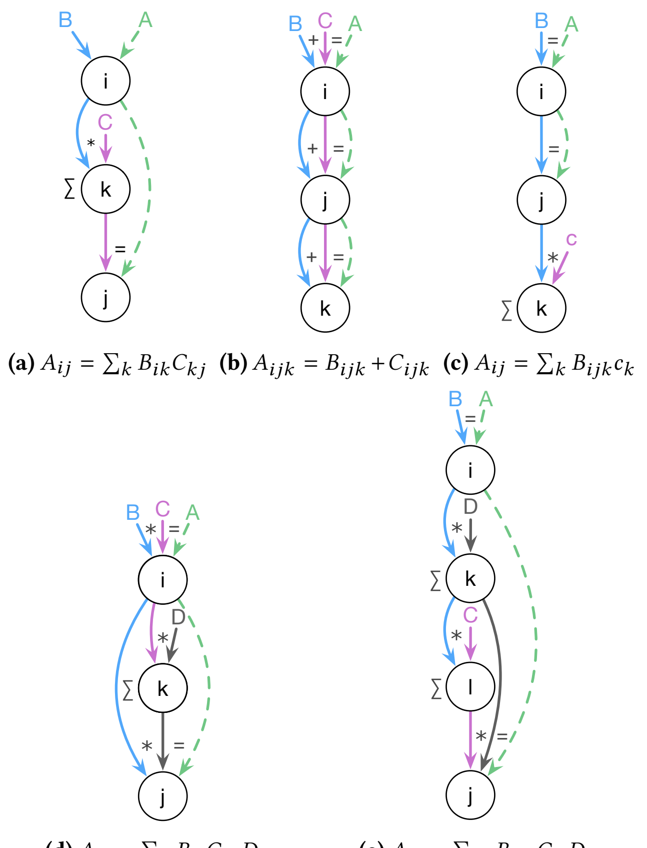
Because matrix multiplication contains the sub-expression B_{ik}, the kernel in Figure 1c iterates over the matrix B with the loops over i (line 1) and k (lines 2–3). The loop over i is dense because the CSR format stores every row. The loop over k, however, is sparse because each row is compressed. 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 the code in Figure 1c.

The kernel is further complicated when the result matrix A is sparse, 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 $\Theta(\text{nnz})$ because the new value must be inserted into the middle of an array. To get the $\Theta(1)$ insertion cost of dense formats, the kernel in Figure 1d introduces a dense workspace. The workspace and accompanying loop transformations is the purpose of the operator split optimization.

A workspace is a temporary tensor that is typically dense with fast insertion and random access. 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 assigning values to the result matrix A, however, it assigns them to 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, which assumes A's CSR index has been pre-assembled. Pre-assembling index structures increases performance when assembly can be moved out of inner loops, which is common in material simulations [26]. We will describe the code to assemble result indices, which keeps track of the nonzero coordinates in the workspace, in Section 6.

3 Tensor Algebra Compilation Background

In this section we will review the iteration graph intermediate representation for tensor algebra compilation [25] that is the target for operator splits. Iteration graphs are constructed from tensor index notation and describe the constraints imposed by sparse tensors on the iteration space of the index



(a) $A_{ij} = \sum_k B_{ik}C_{kj}$ **(b)** $A_{ijk} = B_{ijk} + C_{ijk}$ **(c)** $A_{ij} = \sum_k B_{ijk}c_k$

(d) $A_{ij} = \sum_k B_{ij}C_{ik}D_{kj}$ **(e)** $A_{ij} = \sum_{lk} B_{ikl}C_{lj}D_{kj}$

Figure 2. Five iteration graphs: (a) matrix multiplication, (b) tensor addition, (c) tensor-vector multiplication, (d) sampled dense-dense matrix multiplication (SDDMM), and (e) matricized tensor times Khatri-Rao product (MTTKRP).

variables. Sparse tensors provide an opportunity and a challenge. They store only nonzeros and loops therefore avoid iterating over zeros, but they also enforce a particular iteration order because they encode tensor coordinates hierarchically.

An iteration graph is constructed from tensor index notation, such as $A_{ij} = \sum_{lk} B_{ilk}C_{lj}D_{kj}$ or $a_i = \sum_j B_{ij}c_j + d_i$. In index notation, index variables range over the dimensions they index and computations happen at each point in the iteration space. Index variables are nodes in iteration graphs and each operand access, such as B_{ij}, becomes a path through index variables involved.

Figure 2 shows several iteration graphs, including matrix multiplication, sampled dense-dense matrix multiplication from machine learning [47], and the matricized tensor times Khatri-Rao product used to factorize tensors [6]. The index notation for tensor-vector multiplication is

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

The corresponding iteration graph in Figure 2c has a node for each index variable i, j, and k and a a path for each of

```c
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
  int iB = B1_idx[pB1];
  for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
    int jB = B2_idx[pB2];
    int pA2 = (iB * A2_size) + jB;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
      int kB = B3_idx[pB3];
      int kc = c1_idx[pc1];
      int k = min(kB,kc);
      if (kB == k && kc == k) {
        A[pA2] += B[pB3] * c[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
  }
}
```

Figure 3. Code generated from the iteration graph in Figure 2c when A is dense while B and c are sparse. Each index variable becomes a loop that iterates over the sparse tensor indices of its incoming paths. The k loop iterates over the intersection of the last dimension of B and c.

the three tensor accesses B_{ijk} (blue), c_k (purple), and A_{ij} (stippled green). We draw stippled paths for results. Figure 3 shows code generated from this iteration graph when B and c are sparse. Each index variable node becomes a loop that iterates over the sparse tensor indices belonging to the incoming edges.

Two or more input paths meet at the same index variable when it was used to index into two or more tensors. The iteration space of the tensor dimensions the variable indexes must be merged in the generated code. The index variables are annotated with operators that tell the code generator what kind of merge code to generate. If the tensors are multiplied, then the generated code iterates over the intersection of the indexed tensor dimensions (Figure 5). If they are added, then it iterates over their union (Figure 6). If more than two tensors are indexed by the same index variable, then code is generated to iterate over a mix of intersections and unions of tensor dimensions.

In prior work [25] we described an algorithm to generate code from iteration graphs, including a mechanism called merge lattices to generate code to co-iterate over tensor dimensions. Understanding our workspace optimization does not require understanding the details of the code generation algorithm or merge lattices. We should note, however, that the performance of code that merges sparse tensors may suffer from many conditionals. Code to co-iterate over a combination of a single sparse and one or more dense tensors, on the other hand, does not require conditionals (e.g., Figure 5 and Figure 6). One of the benefits of introducing a workspace is to improve performance by turning sparse-sparse iteration into sparse-dense iteration.

4 Operator Split

Operator splitting is an optimization that applies to a binary operator at an index variable in an index notation expression. It splits the index variable in two and peels off the operator's left sub-expression and stores it to a temporary dense workspace. The original expression is then rewritten to replace the left sub-expression with the workspace. Operator splits have two advantages:

Simplifies merges Splitting a binary operator that has sparse operands in both sub-expressions replaces a sparse-sparse merge with a potentially cheaper sparse-dense merge. By removing sparse-sparse merges we eliminate conditionals and loops and may also remove expensive random inserts into sparse data structures.

Hoists loop invariant computations Splitting an operator may result in the loop for one of the new index variables being emitted in a higher loop nest. That is, it is revealed as loop-invariant and hoisted out of inner loops, which removes redundant computation.

Many important kernels benefit from operator splits, including sparse matrix multiplication, matrix addition, and the matricized tensor times Khatri-Rao product. In this section we describe the optimization with simple examples, but we will explore its application to sophisticated real-world kernels in Section 5.

Splitting an operator in an iteration graph causes the associated index variable i to be divided in two: one for the sub-expression left of the operator (i_{left}) and one for the whole expression (i_{right}). The incoming arrows are divided between the two new index variables. An operator split also introduces a dense workspace that is the result of the first index variable and an operand of the second. In effect, the operator's left sub-expression is stored to the workspace, and the workspace takes the place of the left sub-expression at the second index variable. For example, splitting the multiplication operator in a component-wise multiplication $a_i = b_i c_i$ results in

$$\begin{aligned} w_{i_{\text{left}}} &= b_{i_{\text{left}}} \\ a_{i_{\text{right}}} &= w_{i_{\text{right}}} c_{i_{\text{right}}}. \end{aligned}$$

The rationale for this transformation is that w may be a dense tensor and that the resulting multiplication therefore needs fewer conditional checks, as demonstrated in Figure 5. Furthermore, when the transformation is applied inside a loop nest then it can lead to loop invariant code motion as shown in Section 5.3. Finally, when applied to an assignment it facilitates scattering values into a sparse result. Operator splits can be applied to more than one operator in sequence and each resulting index variable becomes a loop in the generated code.

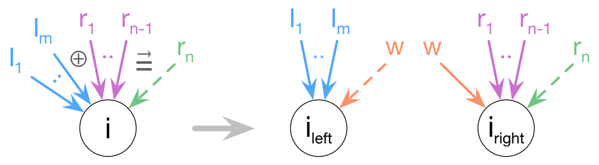
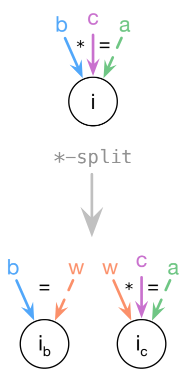


Figure 4. Generic operator split. The arrows to the left of the operator are peeled off to the left index variable, and a workspace is the result of the left and an operand of the right index variable.



```
int pb = b_pos[0];
int pc = c_pos[0];
while (pb < b_pos[1] && pc < c_pos[1]) {
  int ib = b_idx[pb];
  int ic = c_idx[pc];
  int i = min(ib,ic);
  if (ib == i && ic == i) {
    a += b[pb] * c[pc];
  }
  if (ib == i) pb++;
  if (ic == i) pc++;
}
```

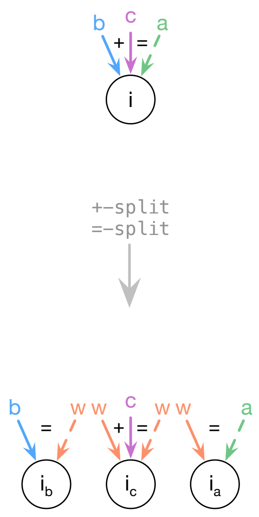
```
for (pb=b_pos[0]; pb<b_pos[1]; pb++) {
  int ib = b_idx[pb];
  w[ib] = b[pb];
}
for (pc=c_pos[0]; pc<c_pos[1]; pc++) {
  int ic = c_idx[pc];
  a += w[ic] * c[pc];
  w[ic] = 0.0;
}
```

Figure 5. Sparse vector inner product $a = \sum_i b_i c_i$. The top shows the iteration graph and associated code that iterates over the intersection of the sparse vectors. The bottom shows the iteration graph and code after splitting the multiplication. The sparse-sparse merge code is replaced with a copy to a dense workspace followed by a sparse-dense merge.

Operator Split Definition: Let $\text{expr}_{\text{left}} \oplus_i \text{expr}_{\text{right}}$ be the index expression at index variable i, where $\oplus$ is any operator including assignment from left to right $\overset{\rightarrow}{=}$. Moreover, let $\text{expr}_{\text{left}}$ include all operands $l_1, \ldots, l_m$ to the left of the operator and let $\text{expr}_{\text{right}}$ include all operands $r_1, \ldots, r_n$ to the right. An operator split of $\oplus_i$ creates two new index variable i_{left} and i_{right}. The index variable i_{left} is given the expression $\text{expr}_{\text{left}} \overset{\rightarrow}{=} w_{i_{\text{left}}}$, and the expression of i_{right} is rewritten to $w_{i_{\text{right}}} \oplus \text{expr}_{\text{right}}$.

An operator split is shown graphically in Figure 4. The index variables i_{left} and i_{right} divide the incoming operator arrows: i_{left} has the arrows from the operator's left side and i_{right} has the arrows from the operator's right side. Furthermore, i_{right} retains the result arrow a, while the result of i_{left} is stored to the workspace w that also replaces the left expression in i_{right}.

Consider two examples, vector product and addition, that show operator splits for iteration graphs with a single variable. In Section 5 we will explore operator splits applied to



```
 1 int pa = a_pos[0];
 2 int pb = b_pos[0];
 3 int pc = c_pos[0];
 4 while (pb < b_pos[1] && pc < c_pos[1]) {
 5   int ib = b_idx[pb];
 6   int ic = c_idx[pc];
 7   int i = min(ib,ic);
 8   if (ib == i && ic == i)
 9     a[pa++] = b[pb] + c[pc];
10   else if (ib == i)
11     a[pa++] = b[pb];
12   else
13     a[pa++] = c[pc];
14   if (ib == i) pb++;
15   if (ic == i) pc++;
16 }
17 while (pb < b_pos[1]) {
18   a[pa++] = b[pb++];
19 }
20 while (pc < c_pos[1]) {
21  a[pa++] = c[pc++];
22 }
```

```
 1 for (pb=b_pos[0]; pb<b_pos[1]; pb++) {
 2   int ib = b_idx[pb];
 3   w[ib] = b[pb];
 4 }
 5 for (pc=c_pos[0]; pc<c_pos[1]; pc++) {
 6   int ic = c_idx[pc];
 7   w[ic] += c[pc];
 8 }
 9 for (pa=a_pos[0]; pa<a_pos[1]; pa++) {
10   int ia = a_idx[pa];
11   a[pa] = w[ia];
12   w[ia] = 0.0;
13 }
```

Figure 6. Sparse vector addition $a_i = b_i + c_i$. The top shows the iteration graph and generated code that iterates over the union of the sparse vectors. The bottom shows the iteration graph and code after splitting the addition and assignment. The sparse merge code is replaced with loops that add each operand to the workspace and one that copies it to the result.

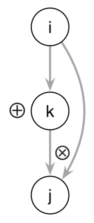
deep iteration graphs that result from higher-dimensional tensor expressions. Figure 5 shows the iteration graphs for a vector inner product before and after splitting the multiplication. The generated code before the split iterates over the intersection of the coordinates in b and c that have nonzero component values. Thus, the loop iterates while both c and b have values left and adds to a if both have a value at a coordinate. After the split the iteration graph has two index variables. The first stores to a dense workspace and the second reads from it. Multiplying a dense and sparse vector does not require merge code since you can iterate over the sparse vector and retrieve values from the dense vector.

The iteration graphs for a vector addition before and after splitting the addition and the assignment operators are shown in Figure 6. Code generated from the iteration graph before splitting has three while loops that together iterate over the union of the operands. The first loop iterates while both b and c have coordinates left and computes results if either have a nonzero at a coordinate; the remaining two loops add the rest of the operand with nonzeros left. We split both the addition and the assignment to show the effect of an assignment split. The resulting code has two loops that store each operand into a workspace and one that copies the nonzeros to a. These loops have fewer conditionals and simpler loop bounds, at the cost of reduced temporal data locality as the reuse distance in w can be large. Which code performs better depends on the machine and on the particular sparsity of the inputs.

The final loop on lines 9–13 in Figure 6 after the split iterates over the result a. This assumes that the index structure of a has been assembled prior to this code executing. Our code generation can emit separate assembly and compute kernels, and in pure compute kernels it assumes that the result index structure has been pre-assembled. We will discuss the assembly code that builds a's index structure in Section 6.

4.1 Precondition

Operator splits do not always apply. Specifically, an operator can be split only if its index variable does not have a reduction variable predecessor in the iteration graph that the operator does not distribute across.

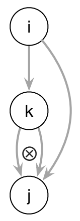


Reduction Precondition: Let j be an index variable with a predecessor reduction variable k with reduction operator $\oplus$. Furthermore, let j and k have a shared predecessor i and let the incoming arrows on j from k and i be merged with the operator $\otimes$. Then j can be split on the operator $\otimes$ if and only if $\otimes$ distributes over $\oplus$ (i.e., $b \otimes (c \oplus d) = (b \otimes c) \oplus (b \otimes d)$).

The reason for the precondition is that after splitting $\otimes$ at j the index variable associated with the arrow from i will no longer be dominated by k. The loop resulting from this j will therefore be hoisted out of the k reduction loop and will be added in only once.

4.2 The Result Tensor as Workspace

If the result is dense then it pays to use it to accumulate the results of both sub-expressions resulting from an operator, instead of introducing a workspace. Thus the result is used as a temporary workspace. This is useful in sparse vector addition when the result is dense. Since the result is dense, there is no need for a temporary workspace. This optimization introduces another precondition to ensure results are not overwritten.



Reuse Precondition: Let j be an index variable that has a predecessor k through two paths that are merged with operator $\otimes_j$. Furthermore, let j and k have a shared predecessor i, and let the arrow from i to k be a result arrow. Then splitting $\otimes_j$ must introduce a new workspace.

This precondition is, for example, not satisfied by the operator split to remove redundant computations in the MTTKRP kernel in Section 5.3. Therefore, a temporary workspace is necessary to optimize that kernel.

4.3 Applying Operator Splits

Operator splitting increases the performance of many important kernels because it removes inserts into sparse results, expensive merge code, and loop invariant code. It does, however, impose costs from constructing, maintaining, and using workspaces. Constructing a workspace requires a `malloc` followed by a `memset` to zero its values and it must be reinitialized between uses. Furthermore, a workspace reduces temporal locality due to the increased reuse distance from storing values to the workspace and later reading them back to store to the result.

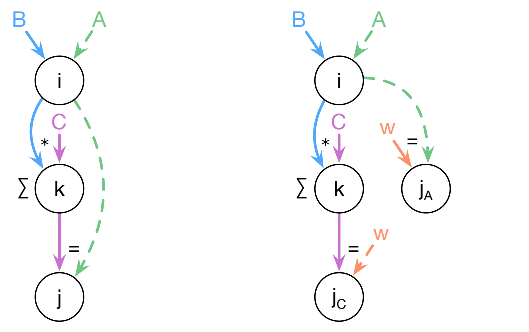
A system design is more flexible if it separates mechanism (what to do) from policy (how to do it) [19, 46]. Performance is a key design criteria in a tensor algebra system. A good system design should therefore separate policy decisions of how to optimize generated code from the mechanisms that carry out the optimization. This paper provides the mechanism for operator splitting.

We imagine many fruitful policy approaches such as user-specified policy, heuristics, mathematical optimization, machine learning, and autotuning. We leave the design of automated policy systems as future work. To facilitate policy research, we see operator splits as a key tensor algebra scheduling construct. The Halide system [39] shows that a scheduling language is effective at separating mechanism from policy. Scheduling languages leave users in control of performance, while freeing them from low level code transformations. The goal, of course, is a fully automated system where users are freed from performance decisions as well. Such a system, however, also profits from a well-design scheduling language because it it lets researchers explore different policy approaches without re-implementing mechanisms. For tensor algebra, a scheduling language should include formats, operator splits, and loop optimizations.

4.4 Dimensionality and Choice of Workspaces

The examples in this paper use vector workspaces. The operator split optimization, however, applies to kernels where higher-dimensional workspaces, such as a matrix or a tensor, are needed. The dimensionality of a workspace is determined by counting the number of index variables above the second index variable after the split, and the sizes of dimensions are determined by the ranges of the index variables.

Furthermore, dense arrays are not the only choice for workspaces; a tensor of any format will do. The format, however, affects the generated code and its performance. Operator splits are often used to remove expensive sparse-sparse merge code, and dense workspaces are attractive because they result in cheaper sparse-dense merges. An alternative is another format with random access such as a hash map. These result in slower execution [36], but uses only memory proportional to the number of nonzeros.



(a) Iteration graph before split **(b)** Iteration graph after split

Figure 7. Matrix multiplication $A_{ij} = \sum_k B_{ik}C_{kj}$, using the linear combination of rows algorithm where all matrices are in the CSR format. Splitting the assignment operator yields Gustavson's algorithm [18], which we showed in Figure 1d.

5 Case Studies

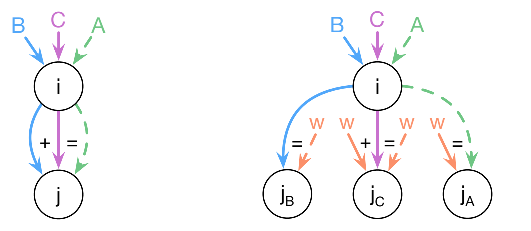
In this section we will study three important linear and tensor algebra expressions that can be optimized with operator splits. The resulting kernels are competitive with hand-optimized kernels from the literature [17, 18, 43]. The optimization, however, generalizes to an uncountable number of kernels that have not been implemented before. We will show one example, MTTKRP with sparse matrices, in Section 5.3.

5.1 Matrix Multiplication

The preferred algorithm for multiplying sparse matrices is to compute the linear combinations of rows or columns [7, 11, 17, 32]. This algorithm was introduced by Gustavson [18], who showed that it is asymptotically superior to computing inner products when the matrices are sparse. Furthermore, both operands and the result are the same format. A sparse inner product algorithm inconveniently needs the first operand to be row major (CSR) and the second column major (CSC).

Figure 7a shows the iteration graph for a linear combination of rows algorithm, where the matrices are stored in the CSR format. The iteration graph has an issue at index variable j. Because the assignment to A at j is dominated by the summation index variable k, the generated code must repeatedly add new values into A. This is expensive when A is sparse due to costly inserts into its sparse data structure.

In Figure 7b, the assignment operator at j has been split to yield two new index variables j_C and j_A. The first index variable j_C accumulates values into a dense workspace w, while j_A copies the nonzero values from the workspace to A. Because the workspace is dense, the merge with C at j_C is trivial: the kernel iterates over C and scatters values into w. Furthermore, the second index variable j_A is not dominated by the summation variable k and values are therefore appended to A.



(a) Iteration graph before split **(b)** Iteration graph after split

Figure 8. Sparse matrix addition $A_{ij} = B_{ij} + C_{ij}$. Splitting the addition and assignment operators removes expensive merge code at the cost of reduced temporal locality. The resulting inner loop is similar to sparse vector addition after an operator split, which we showed in Figure 6.

The code listing in Figure 1d showed the code generated from a matrix multiplication iteration graph where the assignment operator has been split. Each index variable results in a loop, loops generated from index variables connected by an arrow are nested, and loops generated from index variables that share a direct predecessor are sequenced. The loop of j_A copies values from the workspace to A, so it can either iterate over the nonzeros of the workspace or the index structure of A. The loop on lines 10–14 in the code listing iterates over the index structure of A, meaning it must be pre-assembled before this code is executed. The alternative is to emit code that tracks the nonzeros inserted into the workspace, but this is more expensive. It is often more efficient to separate the code that assembles A's index structure from the code that computes its values [18]. For ease of exposition we choose to show pure compute kernels in most code listings. We will, however, discuss code generation for pure assembly and fused assembly-and-compute kernels in Section 6. These kernels cannot assume the results have been pre-assembled and must maintain and iterate over a workspace index.

5.2 Matrix Addition

Sparse matrix addition demonstrates operator splits for addition operators. Sparse additions result in involved code to iterate over the union of the nonzeros of the two operands, as a multi-way merge with three loops [27]. Figure 8a shows the iteration graph for a sparse matrix addition. When the matrices are stored in the CSR format, which is sparse in the second dimension, the compiler must emit code to merge B and C at the j index variable. Such merge code contains many if statements that are expensive on modern processors. Merge code also grows exponential with the number of addition, so if many matrices are added it is necessary to either split the input expression or, better, to use an operator split on the inner index variable so that the outer loop can still be shared.

Splitting the addition and assignment operators at the j index variable introduces a dense row workspace that B and C are in turn are added into, and that is then copied over to A. The resulting code, whose inner loop is similar to Figure 6, has decreased temporal locality, as the workspace reuse distance is can be large, but avoids expensive merges. Whether this results in an overall performance gain depends on the machine and the number of operands that are merged. We show results for one machine in Section 7.

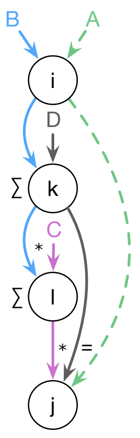
5.3 Matricized Tensor Times Khatri-Rao Product

The matricized tensor times Khatri-Rao product (MTTKRP) is a critical kernel in the alternating least squares algorithms to compute the canonical polyadic decomposition of tensors [21]. It generalizes the singular value decomposition to higher-order tensors, and has applications in data analytics [10], machine learning [37], neuroscience [35], image classification and compression [40], and other fields [28].

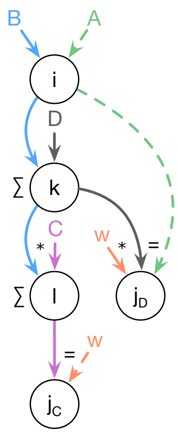
The MTTKRP can be expressed with tensor index notation as $A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$. That is, we multiply a three-dimensional tensor by two matrices in the l and k dimensions. These simultaneous multiplications require four nested loops. Figure 9a shows the iteration graph before optimization, where the matrices are stored row-major. The iteration graph results in four nested loops. The three outermost loops iterate over the sparse data structure of B, while the innermost loop iterates over the range of the j index variable.

After splitting the multiplication operator at j we get the iteration graph in Figure 9b. The index variable j has been split in two. The second variable, j_D, is no longer dominated by l, which means it is evaluated higher up in the resulting loop nest. Furthermore, if the matrices C and D were sparse in the second dimension, the split also removes the need to merge their sparse data structures. The code listing in Figure 9c shows a code diff of the effect of the operator split on the code when the matrices are dense. The code specific to the iteration graph before the $*$-split is colored red, and the code specific to the iteration graph after the split is colored green. Shared code is not colored. The $*$-split results in code where the loop over j, that multiplies B with D, has been lifted out of the l loop, resulting in fewer total multiplication. The drawback is that the workspace reduces temporal locality, as the reuse distance between writing values to it and reading them back can be large. Our evaluation in Section 7 shows that this optimization can result in significant gains on large data sets.

The MTTKRP kernel does two simultaneous matrix multiplications. Like the sparse matrix multiplication kernel in Section 5.1, therefore, it needs to scatter values into the middle of the result matrix A. The reason is that the j and j_D index variables are dominated by reduction variables. If the matrix A is sparse then inserts are expensive, and the code benefits from splitting the assignment at j_D, as shown in Figure 9d. The effect is that values are scattered into a dense



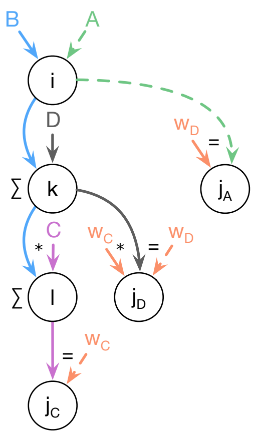
(a) Iteration graph before split



(b) Iteration graph after ∗-split

```
  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
    int i = B1_idx[pB1];

    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
      int k = B2_idx[pB2];

      for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
        int l = B3_idx[pB3];

        for (int jC = 0; jC < C2_size; jC++) {
          int pC2 = (l * C2_size) + jC;
-         int pD2 = (k * D2_size) + jC;
-         int pA2 = (i * A2_size) + jC;
-         A[pA2] += B[pB3] * C[pC2] * D[pD2];
+         w[jC]  += B[pB3] * C[pC2];
        }
      }

+     for (int jD = 0; jD < D2_size; jD++) {
+       int pD2 = (k * D2_size) + jD;
+       int pA2 = (i * A2_size) + jD;
+       A[pA2] += w[jD] * D[pD2];
+       w[jD] = 0.0;
+     }
    }
  }
```

(c) Code diff showing the effect of the ∗-split



(d) Iteration graph after ∗ and = splits

```
  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
    int i = B1_idx[pB1];
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1+1]; pB2++) {
      int k = B2_idx[pB2];
      for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2+1]; pB3++) {
        int l = B3_idx[pB3];
        for (int pC2 = C2_pos[l]; pC2 < C2_pos[l+1]; pC2++) {
          int jC = C2_idx[pC2];
          wC[jC] += B[pB3] * C[pC2];
        }
      }

      for (int pD2 = D2_pos[k]; pD2 < D2_pos[k+1]; pD2++) {
        int jD = D2_idx[pD2];
-       int pA2 = (i * A2_size) + jD;
-       A[pA2] += wC[jD] * D[pD2];
+       wD[jD] += wC[jD] * D[pD2];
      }
      memset(wC, 0, C2_size*sizeof(double));
    }

+   for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
+     int jA = A2_idx[pA2];
+     A[pA2] = wD[jA];
+     wD[jA] = 0.0;
+   }
  }
```

(e) Code diff showing the effect of the =-split

Figure 9. Iteration graphs for the matricized tensor times Khatri-Rao product (MTTKRP) $A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$. Splitting the multiplication at j hoists the multiplication with D out of the inner loop, which removes redundant work. If the matrix A is sparse, then also splitting the assignment introduces a random access workspace that removes the need to insert into A.

workspace with random access, and copied to the result after a full row of the result has been computed. Figure 9e shows a code diff of the effect of making the result matrix A sparse and splitting the assignment operator. Both the code from before the split (red) and the code after (green) assumes the operand matrices C and D are sparse, as opposed to Figure 9c where C and D were dense. As in the sparse matrix multiplication code, the code after assignment split scatters into a dense workspace and, when a full row has been computed, appends the workspace nonzeros to the result.

6 Workspace Assembly

In code listings that compute sparse results, we have so far shown only kernels that compute results without assembling sparse index structures (Figures 1d, 6, and 9e). 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) [18, 20]. The code generation algorithm for iteration graphs can be used to emit

```
 1 A_pos = malloc((m+1)*sizeof(int));
 2 A_idx = malloc(A_idx_size*sizeof(int));
 3
 4 A_pos[0] = 0;
 5 for (int i = 0; i < m; i++) {
 6   for (int pB = B_pos[i]; pB < B_pos[i+1]; pB++) {
 7     int k = B_idx[pB];
 8     for (int pC = C_pos[k]; pC < C_pos[k+1]; pC++) {
 9       int j = C_idx[pC];
10       if (!w[j]) {
11         wlist[w_size++] = j;
12         w[j] = true;
13       }
14     }
15   }
16
17   // Sort row indices
18   sort(wlist, w_size);
19
20   // Make sure A_idx is large enough
21   if (A_idx_size < (A_pos[i]+w_size)) {
22     A_idx_size *= 2;
23     A_idx = realloc(A_idx, A_idx_size*sizeof(int));
24   }
25
26   // Copy workspace indices to A_idx
27   for (int pwlist = 0; pwlist < w_size; pwlist++) {
28     int j = wlist[pwlist];
29     A_idx[A_pos[i] + pwlist] = j;
30     w[j] = false;
31   }
32   A_pos[i+1] = A_pos[i] + w_size;
33   w_size = 0;
34 }
35 A = malloc(A_pos[m]*sizeof(double));
```

Figure 10. A sparse matrix multiplication assembly kernel (the compute kernel is given in Figure 1d). The coordinates of row *i* are inserted into `wlist` on line 11 and copied to `A` on line 29. The array `w` guards against redundant inserts.

either kernel or a kernel that simultaneously assembles the result index structures and computes its values [25].

When generating assembly kernels from iteration graphs, a workspace consists of two arrays that together track its nonzero index structure. The first array `wlist` is a list of coordinates that have been inserted into the workspace, and the second array (`w`) is a boolean array that guards against redundant inserts into the coordinate list.

Figure 10 shows assembly code for sparse matrix multiplication generated from the iteration graph in Figure 7b. It is generated from the same iteration graph as the compute kernel in Figure 1d, so the loop structure is the same except for the loop to copy the workspace to `A` on line 27. In compute kernels, the index structure of `A` must be 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 (`wlist`), on lines 10–13, instead of computing a result, and sorts the index list on line 18 so that the new row of `A` is ordered. Note that the sort is optional and only needed if the result must be ordered. For example, one of the MKL matrix multiplication kernels does not sort. Finally, the assembly kernel allocates memory on lines 1–2, 21–24 (by repeated doubling), and 35.

Table 1. Test matrices and tensors from the SuiteSparse Matrix Collection [12] and the FROSTT Tensor Collection [42].

Tensor	Domain	NNZ	Density
bcsstk17	Structural	428,650	4×10^{-3}
pdb1HYS	Protein data base	4,344,765	3×10^{-3}
rma10	3D CFD	2,329,092	1×10^{-3}
cant	FEM/Cantilever	4,007,383	1×10^{-3}
consph	FEM/Spheres	6,010,480	9×10^{-4}
Facebook	Social Media	737,934	1×10^{-7}
NELL-2	Machine learning	76,879,419	2×10^{-5}
NELL-1	Machine learning	143,599,552	9×10^{-13}

7 Evaluation

In this section, we evaluate the effectiveness of the workspace optimization by comparing performance against hand-written state-of-the-art sparse libraries.

7.1 Methodology

All experiments are run on a dual-socket 12-core/24-thread 2.5 GHz Intel Xeon E5-2680v3 machine with 30 MB of L3 cache per socket, running Ubuntu 14.04.5 LTS. The machine contains 128 GB of memory and runs Linux 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, without counting the first run, which often incurs overheads due to dynamic loading and other first-run overheads. Unless otherwise noted, all experiments are single-threaded.

We evaluate our approach by comparing performance on linear algebra kernels with Eigen [17] and Intel MKL [22] 2018.0. For tensor algebra, we compare against the Matlab Tensor Toolbox [6] and against SPLATT [43], a high-performance C++ library for sparse tensor factorization. We obtained the real-world inputs for the experiments in Sections 7.2 and 7.3 from the SuiteSparse Matrix Collection [12] and the FROSTT Tensor Collection [42] respectively. 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` [24], which places nonzeros randomly to reach a target sparsity. All sparse matrices are in compressed sparse row (CSR) format.

7.2 Sparse Matrix-Matrix Multiplication

Fast sparse matrix multiplication (SpMM) algorithms use workspaces to store intermediate values [18]. We compare our generated workspace algorithm to the SpMM implementations in MKL and Eigen. The approach we described in prior work [25] can in theory handle sparse matrix multiplication by inserting into sparse results. The current implementation, however, does not support this, so we do not compare against its merge-based approach. We compute SpMM with

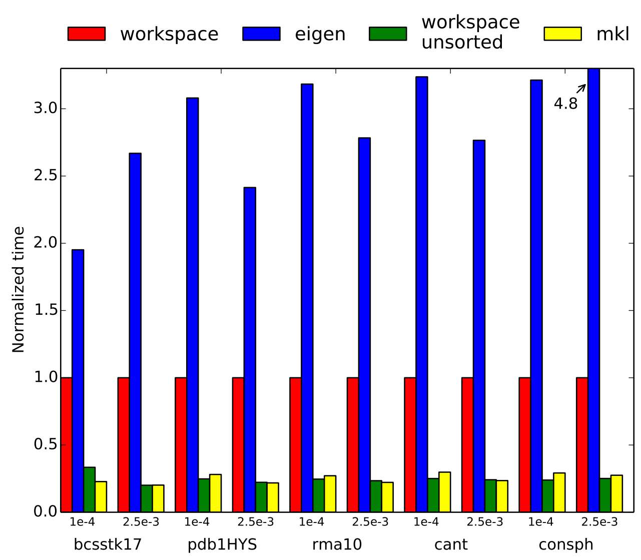


Figure 11. Sparse matrix multiplication results for the matrices in Table 1. We show performance for both sorted and unsorted column entries; Eigen's algorithm sorts them while MKL's `mkl_sparse_spmm` function leaves them unsorted.

Table 2. Breakdown of time, in milliseconds (with 3 significant digits), to multiply the test matrices in Table 1 with a random operand of density 0.0025. Running time is given separately for the workspace assemble and compute kernels, as well as the variant that assembles and computes in one kernel (fused). Times are compared to the total time spent by Eigen and MKL, which do not support separate assembly and compute. For MKL, we use `mkl_sparse_spmm`, which does not sort rows of the output matrix.

	bcsstk17	bcsstk17	rma10	cant	consph
Sorted (ms)					
assembly	331.8	13204	8758	19979	41284
compute	58.07	2398	1742	4184	8480
assembly+compute	389.9	15602	10500	24163	49764
fused	380.5	15141	10279	23398	48106
Eigen	1015	36555	28585	64706	230695
Unsorted (ms)					
assembly	34.79	1510	949.9	2126	4412
compute	58.02	2328	1849	4459	9624
assembly+compute	92.81	3838	2798.9	6585	14036
fused	76.42	3369	2408	5652	12080
MKL	76.78	3300	2279	5507	13231

two operands: a real-world matrix from Table 1 and a synthetic matrix generated with a specific target sparsity, with 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 may appear in any order. Because these two algorithms have very different costs, we implement a workspace variant of each. In addition, we implement two variants of workspace algorithm: one that separates assembly and computation, and one that fuses the two operations.

Figure 11 shows the running times of matrix multiplication for each matrix in Table 1 multiplied by two matrices of different densities (1E-4 and 2.5E-3), using our fused workspace implementation. On average, Eigen is slower than our approach, which generates a variant of Gustavson's matrix multiplication algorithm, by 2.9× and 3.1× respectively for the two sparsity levels. For the unsorted algorithm, we compare against Intel MKL, and find that our performance is essentially the same on average, with our algorithm being up to 22% faster in some cases. Other than one outlier that is 32% slower than MKL, the workspace algorithm is never more than 6% slower than MKL's hand-optimized SpMM implementation.

Table 2 breaks down the running times for the different codes for multiplying with a matrix of density 2.5E-3. Due to sorting, assembly times for the sorted algorithm are quite large; however, the compute time is occasionally faster than the unsorted compute time, due to improved locality when accumulating workspace entries into the result matrix. The fused algorithm is also faster when not using sorting, because otherwise the sort (we use the standard C `qsort`) dominates the time.

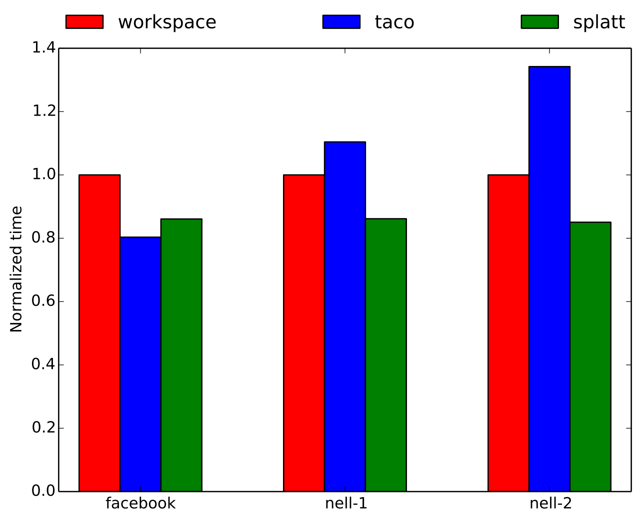


Figure 12. Matricized tensor times Khatri-Rao product (MTTKRP) running times, normalized to the workspace algorithm running time. Only compute times are shown; assembly times are negligible because the outputs are dense.

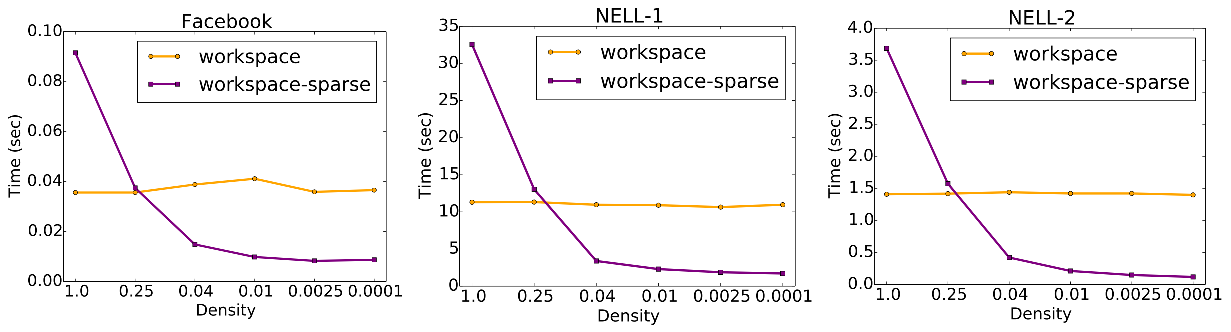


Figure 13. MTTKRP compute time as we vary the density of the matrix operands, for the three test tensors. We compare MTTKRP computed with a workspace when the matrix operands are passed in as dense matrices against an implementation that takes sparse matrices as inputs and outputs a sparse matrix. In all cases, the tensor is passed in using a sparse format.

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. It 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 show only compute times, as the assembly times are negligible because the outputs are dense.

For the NELL-1 and NELL-2 tensors, the workspace algorithm outperforms the merge-based algorithm in `taco` and is within 16% and 12% of the hand-coded performance of SPLATT. On the smaller Facebook dataset, the merge algorithm is faster than both our implementation and SPLATT's. That is, different inputs perform better with different algorithms, which demonstrates the advantage of being able to generate both versions of the algorithm.

7.4 Matricized Tensor Times Khatri-Rao Product with Sparse Matrices

It is useful to support MTTKRP where both the tensor and matrix operands are sparse [41]. If the result is also sparse, then the MTTKRP can be must 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`, although the prior merge-based theory supports it. In this section, we use a workspace implementation of sparse MTTKRP enabled by the operator split optimization. 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.

Which version is faster depends on the density of the sparse operands. Figure 13 shows experiments that compares the compute times for MTTKRP with sparse matrices against MTTKRP with dense matrices, as we vary the density of the randomly generated input matrices. Note that the dense matrix version should have the same performance regardless of sparsity and any variation is likely due to system noise.

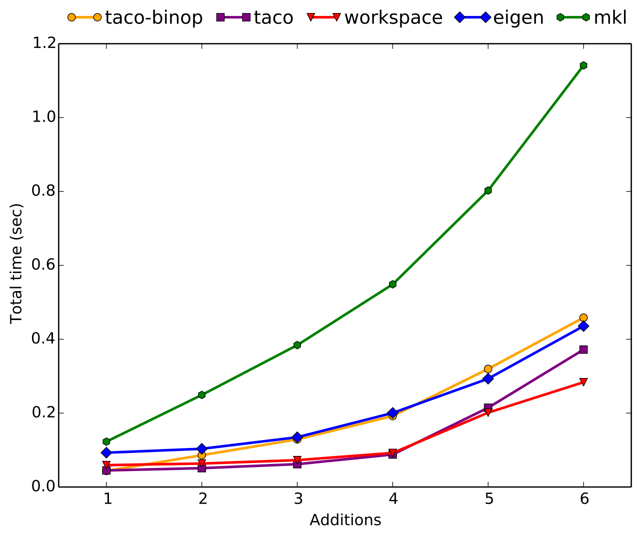


Figure 14. Scaling plot showing the time to assemble and compute n matrix additions with Eigen, MKL, taco binary operations, a single multi-operand taco function, and workspaces. The matrices are described in Table 3.

For each of the tensors, the crossover point is at about 50% 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 that are 99.99% sparse can result in 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 (SpAdd), we show that the algorithm scales as we increase the number of operands. In Figure 14, we compare the workspace algorithm to `taco` using binary operations (as a library would be implemented), `taco` generating a single function for the additions, Intel MKL (using its inspector-executor

Table 3. Breakdown of sparse matrix addition time in ms for adding 7 matrices, for all codes The operands are randomly-generated sparse matrices of density 2.56E-02, 1.68E-03, 2.89E-04, 2.50E-03, 2.92E-03, 2.96E-02, 1.06E-02, respectively.

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

SpAdd implementation), and Eigen. We pre-generate k matrices with the target sparsities chosen uniformly randomly from the range $1E^{-4}$–0.04 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 the restriction that they perform 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 SpAdd. 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. Table 3 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 assemble 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 SpMM.

8 Related Work

Related work is divided into work on tensor algebra compilation, work on manual workspace optimizations of matrix and tensor kernels, and work on general loop optimization.

There have been much work on optimizing dense matrix and tensor computations [3, 23, 34, 45]. Researchers have also worked on compilation and code generation of sparse matrix computations, starting with the work of Bik and Wijshoff [8] and the Bernoulli system [30]. Recently, we proposed a tensor algebra compilation theory built on an intermediate representation called iteration graphs [25] that are constructed from tensor index notation. We gave an algorithm to generate code, but did not introduce any optimization passes. The operator split optimization presented in this paper is, to the best of our knowledge, the first optimization on iteration graphs. As we have shown, it improves the performance of the kernels generated from many iteration graphs by removing merges, hoisting loop invariant code, and handling scattering to sparse result tensors.

The first use of dense workspaces for sparse matrix computations is Gustavson's sparse matrix multiplication implementation, that we recreate with an operator split in Figure 7 to produce the code in and Figure 1d [18]. A workspace used for accumulating temporary values is referred to as an expanded real accumulator in [38] and as an abstract sparse accumulator data structure in [16]. Dense workspaces and blocking are used to produce fast parallel code by Patwary et al. [36]. 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 $A^T x$ [9]. Finally, Smith et al. uses a workspace to hoist loop-invariant code in their implementation of MTTKRP in the SPLATT library [43]. We re-create this optimization with an operator split in Figure 9b and show the resulting source code in Figure 9c.

One use of operator splitting 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 [4]. Recently, researchers have found new opportunities for removing redundancy in loops by taking advantage of high-level algebraic knowledge [13]. The operator split optimization is done on the high-level iteration graph intermediate representation that expose sparse dependencies. Since it is applied prior to code generation, it avoids the need to analyze low-level code. It can therefore apply loop invariant code motion to loops that implement sparse computations with many branches.

9 Conclusion

This paper presented the first compiler optimization on the iteration graph intermediate representation for tensor computations. A single operation—operator split—generalizes several manual optimizations described in the literature on sparse matrix and sparse tensor codes. These optimizations remove expensive merge code, avoid the need to scatter into sparse results, and hoist partial sparse tensor computations out of inner loops. Future work includes exploring tradeoffs for different types of workspaces and automating the decision of when to split.

References

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

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

[3] 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.

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

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

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

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

[8] 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.

[9] 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.

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

[11] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. Vol. 2. Siam.

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

[13] Yufei Ding and Xipeng Shen. 2017. GLORE: Generalized Loop Redundancy Elimination upon LER-Notation. (2017).

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

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

[16] 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.

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

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

[19] Per Brinch Hansen. 1970. The Nucleus of a Multiprogramming System. *Commun. ACM* 13, 4 (April 1970), 238–241. https://doi.org/10.1145/362258.362278

[20] Michael T Heath, Esmond Ng, and Barry W Peyton. 1991. Parallel algorithms for sparse linear systems. *SIAM review* 33, 3 (1991), 420–460.

[21] 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.

[22] 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.

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

[24] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. taco: a tool to generate tensor algebra kernels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 943–948.

[25] 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

[26] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David IW Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M Kaufman, Gurtej Kanwar, Wojciech Matusik, et al. 2016. Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG)* 35, 2 (2016), 20.

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

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

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

[30] 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.

[31] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

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

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

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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).

[40] 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.

[41] 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.

[42] 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/

[43] 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.

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

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

[46] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM* 17, 6 (June 1974), 337–345. https://doi.org/10.1145/355616.364017

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