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).
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 data structure. It consists of two index arrays \( B_{\text{pos}} \) and \( B_{\text{idx}} \) and a value array \( B \). The array \( B_{\text{idx}} \) contains the column coordinates of nonzero values in corresponding positions in \( B \). The array \( B_{\text{pos}} \) stores the position of the first column coordinate of each row in \( B_{\text{idx}} \) and a sentinel with the number of nonzeros in the matrix \( \text{nnz} \). Thus, contiguous values in \( B_{\text{pos}} \) store the beginning and end [inclusive-exclusive] of a row in the arrays \( B_{\text{idx}} \) and \( B \). For example, the column coordinates of the third row are stored in \( B_{\text{idx}} \) at positions \( B_{\text{pos}}[2], B_{\text{pos}}[3] \). 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 computes 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 \( i \)th row, the \( k \) loop iterates over \( [B_{\text{pos}}[i], B_{\text{pos}}[i+1]] \) in \( B_{\text{idx}} \). 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:

```cpp
1 Format CSR((dense, sparse));
2 TensorVar A(CSR), B(CSR), C(CSR);
3 IndexVar i, k, j;
4 IndexExpr mul = B(i,k) * C(k,j);
5 A(i,j) = sum(k, mul);
```
Sparse Tensor Algebra Optimization with Workspaces

Figure 1. Figure (a) shows a sparse matrix and (b) its compressed data structure. Furthermore, 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.

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 tensor operations, it is unsuitable as a compiler IR. The reason is that it does not specify the order of execution or temporaries 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:

\[
\forall_{ikj} A_{ij} += B_{ik} C_{kj}
\]

The assignment statement \( A_{ij} += B_{ik} C_{kj} \) 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 \( \forall_i \forall_j \forall_k \), 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:

\[
\forall_{ikj} A_{ij} += B_{ik} C_{kj}
\]
∀i (Vj Ai,j = wij) where

(∀k ∈ K
  wk = Dk,j;
  ∀j ∈ J
    Aj,j = wij)

(∀k ∈ K
  wk + Bj,kCk,j;
  ∀j ∈ J
    Aj,j = wij)

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:

∀i (Vj Ai,j = wij) where

(∀j ∈ J
  wj = Dij;
  ∀k ∈ K
    wk = Dk,j;
  ∀j ∈ J
    Aj,j = wij)

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

Let S_A^w be S_A where E has been replaced by the access expression wj where w is a fresh tensor variable.

In S, replace S_A with S_A^w where (wj ⊕= E).

Let S_w be this where statement.

while S_w is contained in some forall statement Vj S_w do
  if j is used in both sides of S_w and j ∉ I then
    Move Vj into both sides of S_w.
  else if j is used only in the left side of S_w then
    Move Vj into the left side of S_w.
  else if j is used only in the right side of S_w then
    Move Vj 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 ⊕. The dimensionality of the resulting workspace is the number.
Sparse Tensor Algebra Optimization with Workspaces

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:

$$\forall_i a_i = b_i + c_i \implies (\forall_i a_i = b_i; \forall_i a_i = c_i).$$

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 precondition 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 over-written by its use as a workspace. For example, this precondition is not satisfied by the second workspace optimization to the MTTKRKP 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

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

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

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

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

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

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 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 encounters 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 $v_{ijk} A_{ij} += B_{ijk} c_k$, where $B$ and $c$ are sparse.

![Figure 5. Iteration graph and sparse code of a tensor-vector multiplication $v_{ijk} A_{ij} += B_{ijk} c_k$, where $B$ and $c$ are sparse.](image)

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...
Sparse Tensor Algebra Optimization with Workspaces

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 $w_Dk_j$. The first transformation removes the edge between $jD$ 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.

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 $A_{ij} = \sum_{kl} B_{ikl}C_{lj}$. We multiply a 3-order tensor by two matrices in the 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

\[ \forall_{ikl} A_{ij} += B_{ikl}C_{lj}D_{kj}, \]

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 $w_Dk_j$: \[ \forall_{ik} A_{ij} += w_Dk_j D_{kj} \] where $\forall_{ij} w_j += B_{ikl}C_{lj}$. 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 that shows the effect on the compiled code of applying the transformation to pre-compute $B_{ikl}C_{lj}$ into the workspace $w_Dk_j$ is shown in Figure 7. 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 been replaced by $j_C$ on the result path (green dashed) have been replaced by $j_C$.

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

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

<table>
<thead>
<tr>
<th>Tensor</th>
<th>Domain</th>
<th>NNZ</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcsstk17</td>
<td>Structural</td>
<td>428,650</td>
<td>4E-3</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>Protein data base</td>
<td>4,344,765</td>
<td>3E-3</td>
</tr>
<tr>
<td>rma10</td>
<td>3D CFD</td>
<td>2,329,092</td>
<td>1E-3</td>
</tr>
<tr>
<td>cant</td>
<td>FEM/Cantilever</td>
<td>4,007,383</td>
<td>1E-3</td>
</tr>
<tr>
<td>consph</td>
<td>FEM/Spheres</td>
<td>6,010,480</td>
<td>9E-4</td>
</tr>
<tr>
<td>cop20k</td>
<td>FEM/Accelerator</td>
<td>2,624,331</td>
<td>2E-4</td>
</tr>
<tr>
<td>ships1c</td>
<td>FEM</td>
<td>3,568,176</td>
<td>2E-4</td>
</tr>
<tr>
<td>scircuit</td>
<td>Circuit</td>
<td>958,936</td>
<td>3E-5</td>
</tr>
<tr>
<td>mac-econ</td>
<td>Economics</td>
<td>1,273,389</td>
<td>9E-5</td>
</tr>
<tr>
<td>pwtk</td>
<td>Wind tunnel</td>
<td>11,524,432</td>
<td>2E-4</td>
</tr>
<tr>
<td>webbase-1M</td>
<td>Web connectivity</td>
<td>3,105,536</td>
<td>3E-6</td>
</tr>
<tr>
<td>Facebook</td>
<td>Social Media</td>
<td>737,934</td>
<td>1E-7</td>
</tr>
<tr>
<td>NELL-2</td>
<td>Machine learning</td>
<td>76,879,419</td>
<td>2E-5</td>
</tr>
<tr>
<td>NELL-1</td>
<td>Machine learning</td>
<td>143,599,532</td>
<td>9E-13</td>
</tr>
</tbody>
</table>

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.\footnote{According to MKL documentation, its sorted algorithms are deprecated and should not be used.} 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\footnote{as of Git revision bfe8bb6} does not generate sparse matrix multiplication, because

---

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.
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 4x and 3.6x 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 4.5–11x 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
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 2.8x 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 $A^T x$ [5]. Finally, Smith et al.
Sparse Tensor Algebra Optimization with Workspaces

uses a workspace to hoist loop-invariant code in their implementation of MTTKR-P 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


