Compilation of Modular and General Sparse Workspaces

GENGHAN ZHANG, Stanford University, USA
OLIVIA HSU, Stanford University, USA
FREDRIK KJOLSTAD, Stanford University, USA

Recent years have seen considerable work on compiling sparse tensor algebra expressions. This paper addresses a shortcoming in that work, namely how to generate efficient code (in time and space) that scatters values into a sparse result tensor. We address this shortcoming through a compiler design that generates code that uses sparse intermediate tensors (sparse workspaces) as efficient adapters between compute code that scatters and result tensors that do not support random insertion. Our compiler automatically detects sparse scattering behavior in tensor expressions and inserts necessary intermediate workspace tensors. We present an algorithm template for workspace insertion that is the backbone of our code generation algorithm. Our algorithm template is modular by design, supporting sparse workspaces that span multiple user-defined implementations. Our evaluation shows that sparse workspaces can be up to 27.12× faster than the dense workspaces of prior work. On the other hand, dense workspaces can be up to 7.58× faster than the sparse workspaces generated by our compiler in other situations, which motivates our compiler design that supports both. Our compiler produces sequential code that is competitive with hand-optimized linear and tensor algebra libraries on the expressions they support, but that generalizes to any other expression. Sparse workspaces are also more memory efficient than dense workspaces as they compress away zeros. This compression can asymptotically decrease memory usage, enabling tensor computations on data that would otherwise run out of memory.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Source code generation**.

Additional Key Words and Phrases: sparse tensor algebra, compilation, sparse workspaces, code composition

ACM Reference Format:
Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. *Proc. ACM Program. Lang.* 8, PLDI, Article 196 (June 2024), 26 pages. https://doi.org/10.1145/3656426

1 INTRODUCTION

Sparse tensor algebra is an important class of computation used in various applications [8, 34, 36, 40, 41, 48]. It generalizes linear algebra to higher-order tensors, where the tensors may be dense or sparse. Domain-specific sparse tensor algebra compilers [39, 74, 83, 87] automatically generate and optimize sparse tensor algebra code. These compilers are becoming more prevalent because they can generate codes for the large combination of tensor algebra expressions, compressed data structures, optimizations, and hardware backends that are not supported by libraries [30, 79, 81, 85].

However, there is a hole in the above sparse compiler work: the sparse scattering problem. Sparse scattering happens when a sparse result tensor is written to in an arbitrary order. This is a common problem in sparse tensor algebra [49, 58, 82]. Figure 1 shows a concrete example of sparse scattering. In this example, the tensor component (or element) needs to be inserted in front of components that have already been placed. Specifically, the tensor component generated in state (d) has coordinates that are lexicographically smaller than the coordinates from states (a)–(c). However, the result

Authors' addresses: Genghan Zhang, Stanford University, USA, zgh23@stanford.edu; Olivia Hsu, Stanford University, USA, owhsu@stanford.edu; Fredrik Kjolstad, Stanford University, USA, kjolstad@stanford.edu.

ACM 2475-1421/2024/6-ART196
https://doi.org/10.1145/3656426

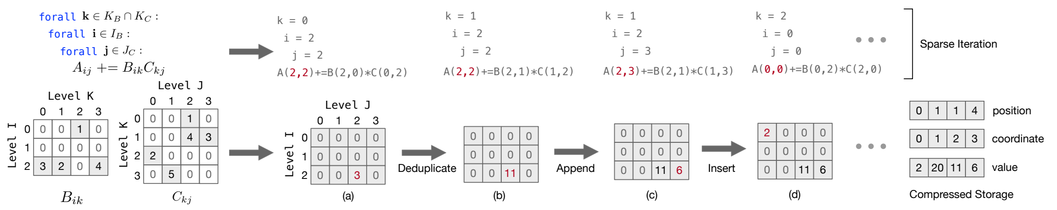


Fig. 1. A second-order dense workspace for outer-product matrix multiplication (SpGEMM). The above for-loop pseudo codes show the sparse iterations that generate tensor components. Red numbers represent newly generated coordinates and values. The workspace must support three behaviors: deduplicating (a ⟶ b), appending (b ⟶ c), and inserting (c ⟶ d). The computation utilizes a workspace since the final compressed data structures do not support insertion. Furthermore, the result storage should be compressed for memory efficiency since the final output has only four values.

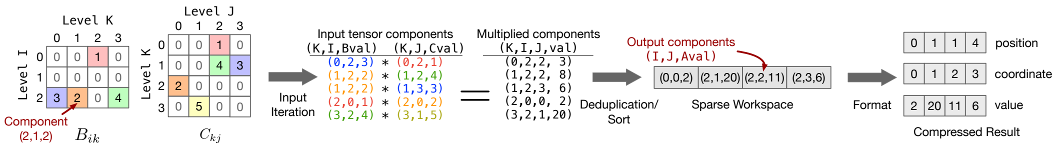


Fig. 2. A second-order sparse workspace for the outer-product SpGEMM in Figure 1. The colored nonzero components of the input tensors show a correspondence to their respective input tensor components. In a second-order workspace, each `(I,J,val)` tensor component is indexed by two variables `I` and `J`.

matrix is stored in a compressed data structure, which does not permit efficient insertion into this location. Therefore, a temporary tensor is necessary as an adapter between the computed matrix components and the result tensor storage. Such a temporary is called a workspace.

Figure 1 and Figure 2 show examples of a dense and a sparse workspace, respectively. A dense workspace holds the temporary values in a dense array whereas a sparse workspace stores temporary values in compressed data structures. In order to store a tensor component (both coordinates and the value) into any workspace, the value must first be calculated from the inputs. The value is then summed with the existing value in the workspace during the deduplication step. Then, the deduplicated value is either appended to the end of or inserted into the middle of the workspace data structure. An example of this process is shown in Figure 1. In the case of a sparse workspace (as in Figure 2), it is common to only append values to the compressed data structure (without insertion) for efficiency. In order to ensure that the values end up in the correct output order, sparse workspace algorithms include a sorting step.

A sparse workspace is essential when sparse scattering into high-order tensors because dense higher-order storage would require too much memory. Sparse tensors are often asymptotically sparse, which empirically implies that less than 1% of values are nonzero [37]. For example, multiplying the "marine1" matrix (from chemical oceanography [42]) by itself transposed would require a dense workspace that is 5132× larger than the sparse workspace. Beyond SpGEMM, it is common to have higher-order workspaces when the input and output tensors are higher-order, as in computation like sparse convolution [82].

We would like to support sparse workspaces in sparse compilers, but there are two challenges:

(1) The compiler should be modular to allow users to automatically plug in and combine various optimization strategies. Workspace policies are diverse [8, 12, 18], so modularity is essential for generating code that is competitive with libraries.
(2) The compiler should be sufficiently general so as to produce code for any expressions with sparse scattering. That means the sparse workspaces generated by the compiler must handle tensors of any order and with any compression format.

To address these challenges, we propose an algorithm template for sparse workspaces and a compiler that integrates hand-written implementations of the template functions to generate correct and efficient code. Generating code by combining templated code with filled-in, user-written functions (also called codelets or stencils) has been used throughout history. This idea of *code composition* through templates is widely used in compilers for domain-specific languages (DSLs) [73]. Specific examples include compiling relational algebra queries [55], FFTs [23], and more recently tensor algebra [7, 32] and neural networks [4, 13]. Beyond DSLs, general programming systems have also benefited from similar compositional methods through techniques like template JIT [21, 22, 61, 80]. We are, however, the first to propose a modular template-based approach for compiling sparse tensor algebra expressions with scattering behavior.

For the algorithm template, we design a modular representation of sparse workspaces expressive enough to describe several prior, efficient sparse workspace policies. For the compiler, we incorporate our representation into the TACO system [39] to provide general support for generating sequential implementations of sparse tensor algebra expressions with sparse workspaces. We design user interfaces at different levels of our system to express new policies. Our contributions are:

- An analysis framework that categorizes sparse tensor algebra expressions with respect to how they assemble the result.
- An algorithm template for sparse workspace generation. The algorithm generalizes to workspaces implemented with various compressed data structures and optimization policies.
- An automatic workspace insertion algorithm that transforms expressions to include those workspaces that are necessary for correctness.
- Extensions to the TACO programming model and intermediate representation to generate code for expressions with sparse workspaces.

We evaluate these contributions by comparing them against current dense workspace techniques. Our evaluation shows that sparse and dense workspaces do not dominate each other but are useful in different situations, motivating the need for both approaches. Specifically, dense workspaces can perform up to 7.58× faster than sparse workspaces, whereas sparse workspaces can perform up to 27.12× faster than dense ones, depending on the input data. Our compiler can generate general tensor algebra codes with both sparse and dense workspaces while still producing code competitive with hand-optimized linear and tensor algebra libraries. Furthermore, our compiler produces code with a 3.6× improvement in memory footprint on average (geomean) when compared to dense workspaces that fit in our machine's memory. The sparse workspace code generated by our compiler is not limited by the machine's memory for any input data, but the dense workspace is unable to fit in memory for 10 out of the 60 input matrices run. Therefore, our compiler scales better both in performance and memory footprint for higher-order and larger data sizes. Though our approach generates sequential sparse workspace code, it provides a foundation for developing a code generator that also supports parallel sparse workspace codes.

2 SPARSE TENSOR ALGEBRA EXPRESSION TAXONOMY

We introduce an analysis framework to identify when sparse scattering occurs in sparse tensor algebra expressions. Sparse scattering can be classified based on the relationship between the tensor access order and the expression loop order. These orderings, defined in Section 2.1, determine how input tensors are accessed and how the result tensor is assembled. Based on these orders, we classify sparse tensor algebra expressions along two axes (Section 2.2) and place our work in the context of prior work by showing how they lack the ability to handle sparse scattering (Section 2.3).

Index Variable	i	*Constant*	c	*Tensor*	$\mathcal{T}$
Access	a	$::=$	$\mathcal{T}_{i*}$		
Expression	e	$::=$	$a \mid c \mid e+e \mid e \cdot e \mid \ldots$		
Statement	S	$::=$	$\forall_{i*}\ S \mid a = e \mid a \mathrel{+}= e \mid \ldots$		

Fig. 3. A simplified concrete index notation (CIN) syntax with no scheduling relationships.

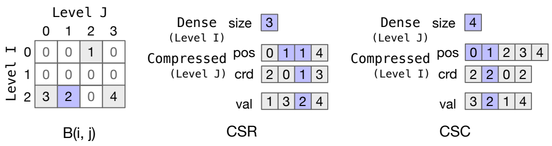


Fig. 4. Two example tensor level formats for compressed sparse row (CSR) and compressed sparse column (CSC).

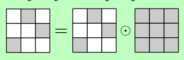
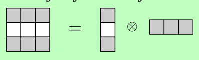
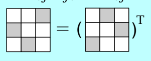
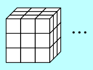
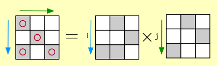
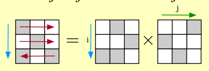
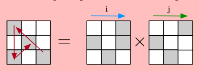
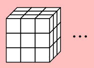
Computation \ Ordering	0	1	2	n
Appending	(a) $\forall_{ij} A_{ij} = B_{ij} C_{ij}$  $B:\{i,j\} \to \{i,j\}$ $C:\{i,j\} \to \{i,j\}$	(b) $\forall_{ij} A_{ij} = B_i C_j$  $A:\{i,j\} \to \{i,j\}$	(c) $\forall_{ij} A_{ji} = B_{ij}$  $A:\{j,i\} \to \{j,i\}$ $B:\{i,j\} \to \{i,j\}$	(d) 
Scattering	(e) $\forall_{ijk} A_{ij} \mathrel{+}= B_{ik} C_{kj}$  $B:\{i,k\} \to \{i,k\}$ $C:\{k,j\} \to \{j,k\}$	(f) $\forall_{ikj} A_{ij} \mathrel{+}= B_{ik} C_{kj}$  $B:\{i,k\} \to \{i,k\}$ $C:\{k,j\} \to \{k,j\}$	(g) $\forall_{kij} A_{ij} \mathrel{+}= B_{ik} C_{kj}$  $B:\{i,k\} \to \{k,i\}$ $C:\{k,j\} \to \{k,j\}$	(h) 

Fig. 5. Sparse tensor algebra expressions classified by computation and ordering. Blue and green arrows show the loop order and red lines show the result assembly order. Tensors' index variables encode access order.

2.1 Index Variable Orderings

Sparse tensor algebra expressions, given in concrete index notation (CIN) [38], are composed of tensors, index variables, and forall nodes. CIN is a loop-based intermediate representation in which physical tensor representations are abstracted away. For the tensor abstraction, tensors are stored level by level. Each level describes the coordinates of one tensor mode and can be materialized as a data structure in some format. Figure 3 gives the syntax of the core parts of CIN in this paper.

Sparse tensor algebra expressions along with their tensor formats have two types of ordering properties: access orders and loop orders. An access order describes how an individual tensor *needs* to be accessed (read) and assembled (written to). The loop order, on the other hand, determines how each tensor in an expression *is actually* accessed and assembled. The loop order is directly defined as the order of loop index variables—index variables next to ∀ nodes in Figure 3.

The access order defines the order of the index variables that access the physical storage of a tensor. It is generated by an `AccessMap` function that maps the index variables used to index (access) a tensor to the storage levels of its format. The access index variables are the index variables inside an Access in Figure 3. The *AccessMap* builds upon the established level formats such as *Dense* and *Compressed* in Kjolstad et al. [39] and the coordinate mapping described in Chou et al. [15].

Let us consider outer-product SpGEMM $A_{ij} = \sum_k B_{ik} C_{kj}$ where A and C are stored in the compressed sparse rows format (CSR) and B is stored in compressed sparse columns (CSC) (shown in Figure 4). In CIN, this is expressed as $\forall_{kij}\ A_{ij} \mathrel{+}= B_{ik} C_{kj}$ with the following tensor AccessMaps:

(1) $\text{AccessMap}_A(\{i,j\}, \{Dense, Compressed\}) = \{i,j\}$,
(2) $\text{AccessMap}_B(\{i,k\}, \{Dense, Compressed\}) = \{k,i\}$, and
(3) $\text{AccessMap}_C(\{k,j\}, \{Dense, Compressed\}) = \{k,j\}$.

Therefore, the access orders for A, B and C are $i \longrightarrow j$, $k \longrightarrow i$, and $k \longrightarrow j$ respectively, and the loop order for the expression is $k \longrightarrow i \longrightarrow j$.

2.2 Classification

We classify sparse tensor algebra expressions based on the computation and ordering of the result coordinates as shown in Figure 5. The computation axis describes whether each resulting value corresponds to a single value computed from the operands or whether it corresponds to a combination of many computed values (e.g., a sum). The ordering axis describes how the result coordinates are generated, and to what extent the generation order matches the access order of the output tensor. Intuitively, the order of input sparse iteration with respect to the order of output access decides the number of dimensions we need in the intermediate workspace data structure. For example, an inner-product matrix multiplication requires only a scalar (order 0) temporary, while an outer-product matrix multiplication requires a matrix (order 2) temporary.

The result coordinate calculation is *appending* if the pattern of nonzeros of the result is the same as the input iteration space and *scattering* otherwise. As shown in Figure 5(a), the element-wise multiplication of a sparse and dense matrix is appending because the sparse result matrix A has the same coordinates as the sparse input matrix B. Tensor transpositions as in Figure 5(c) are also appending because the input coordinates do not require an intersection or union to compute the result coordinates even though the result coordinates are transposed.

The result coordinates likely needs to be assembled in a *sparse* way if the result coordinate ordering can not be narrowed to a first-order (vector) or zero-order (scalar) tensor as the loop proceeds, due to the large worst-case memory cost of storing a sparse matrix or tensor in a dense data structure. In other words, the assembly is sparse if the loop order mismatches with the output access order at a position greater than one. As shown in Figure 5(f), the loop order of the row-wise SpGEMM is $i \longrightarrow k \longrightarrow j$, and the output access order is $i \longrightarrow j$. These two orders only mismatch on index k with k at the first position (from the inner-most index), so we classify the expression as a first-order *dense* scattering (the yellow area in Figure 5). On the contrary, the loop order of the outer-product SpGEMM in Figure 5(g) is $k \longrightarrow i \longrightarrow j$, which mismatches with the output access order with k at the second position. Therefore, we classify the expression as second-order *sparse* scattering (the red area in Figure 5).

In general, ordering is determined by the position of the first index variable in the loop order that does not match the access order. Given a loop order $\mathcal{L} = i_M \longrightarrow i_{M-1}... \longrightarrow i_1$, and the output tensor's access order $\mathcal{A} = j_N \longrightarrow j_{N-1}... \longrightarrow j_1$, there are two important positions where the orders mismatch. The first position occurs during tensor transpositions, where p_1 is the position of the first index variable from the left in $\mathcal{A}$ that mismatches with $\mathcal{L}$. The second is p_2, the position of the first non-access index variable i in $\mathcal{L}$. If i is before the first access index variable j_1, then $p_2 = 0$ because output tensor components are accumulated scalar by scalar. If i is after the last access index variable j_N, then $p_2 = N$ because all components in the output tensor are accessed multiple times. Otherwise, the p_2 equals the i's position in the middle of $\mathcal{L}$ between j_N and j_1. The result coordinates' ordering equals $max(N + 1 - p_1, p_2)$. The materialized workspace cannot have fewer orders than this ordering. We provide a more detailed algorithm that determines the required order of a workspace in Section 6.2.

2.3 Limitation of Prior Work

As shown in Table 1, prior sparse compilers [38, 39, 83, 86] cannot generate code for expressions with sparse scattering behavior. Although compilers can generate efficient co-iteration on sparse input tensors [74, 87], they often assume the output compression format is known beforehand as dense or identical to one of the compressed inputs [83, 86]. Compilers that generate code with dense workspaces [9, 10, 38, 44, 62] can assign a temporary dense array to hold values generated from the input co-iteration. However, they either only operate on linear algebra [10] or consume too much memory when densifying a sparse higher-order tensor into the temporary [38]. The TACO body of work provides format conversions [15] that can handle assignment expressions where the result tensor has the same elements as the input but with varying formats. However, it cannot process dynamic, out-of-order nonzeros generated by sparse iteration with scattering. As

Table 1. Output tensor support in prior work. The ✓ denotes full support and ✗ denotes no support.

Sparse Tensor Algebra Compilers	Output Appending		Output Scattering		
	Dense	Sparse	Dense	Sparse	
				Dense workspace	Sparse workspace
SparseTIR [83]	✓	✗	✗	✗	✗
Sparse Polyhedral Framework [86]	✓	✗	✗	✗	✗
MLIR Sparse Dialect [9]	✓	✗	✓	✓	✗
TACO with dense workspaces [38]	✓	✗	✓	✓	✗
TACO with format conversion [15]	✓	✓	✗	✗	✗
Our work	✓	✓	✓	✓	✓

such, our work is the first sparse tensor algebra compiler that takes in dynamic components from input tensors and scatters them into tensors of any format.

3 OVERVIEW

We implemented the new compiler techniques for sparse workspaces as an extension to the open-source tensor compiler TACO [39], but these technique can also be used in other sparse tensor compilers [9, 52, 83]. Figure 6 gives an overview of our new compiler with sparse workspaces. It takes as input an expression in tensor index notation (or Einsum Notation), a format language [14] that encodes the AccessMap, and a scheduling language [65] that incorporates sparse workspaces. Our compiler combines these three input languages into the CIN intermediate representation.

We propose the insert-sort-merge (ISM) algorithmic template for generating sparse workspace code that can be inserted into generated sparse tensor algebra computation loop nests. The ISM algorithm is the algorithmic backbone that gives us a straightforward way to generate modular sparse workspace code (Section 4). Whenever a CIN expression contains a sparse workspace tensor, our compiler lowers it into dense and sparse loops with the ISM template inserted. The ISM template only defines abstract memory pools and function interfaces, with holes for different sparse workspace building blocks. During code generation, our compiler fills in those holes with user-defined workspace implementations (Section 6.3).

We explore several concrete sparse workspace policies in Section 5 that are compatible with the ISM template. These are example user-defined workspace policies that materialize the ISM template during code generation, but many more are possible. Users provide their ISM template function implementations as input into the format language of our compiler (Section 6.1).

Our compiler technique operates across various stages of compilation and bridges prior work on sparse iteration [30, 39] with prior work on format conversions [15] using the concepts from the ISM template. Apart from the format language, we extend the scheduling language to include sparse workspaces. For user productivity, we also automate the insertion of sparse workspaces to

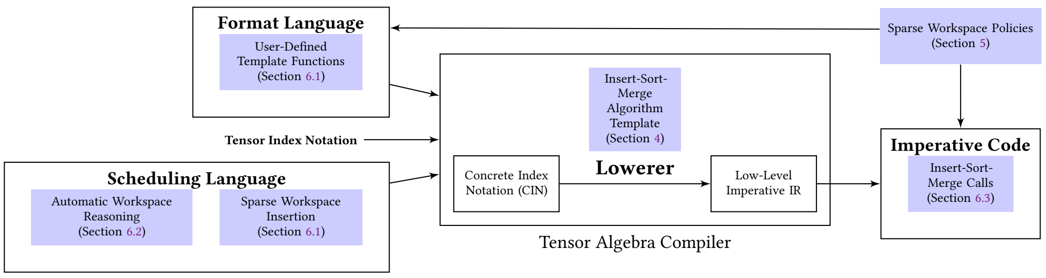

Format Language
User-Defined Template Functions (Section 6.1)
Sparse Workspace Policies (Section 5)
Insert-Sort-Merge Algorithm Template (Section 4)
Tensor Index Notation
Scheduling Language
Automatic Workspace Reasoning (Section 6.2)
Sparse Workspace Insertion (Section 6.1)
Concrete Index Notation (CIN)
Lowerer
Low-Level Imperative IR
Imperative Code
Insert-Sort-Merge Calls (Section 6.3)
Tensor Algebra Compiler

Fig. 6. System Overview. Blue components denote new contributions of this work.

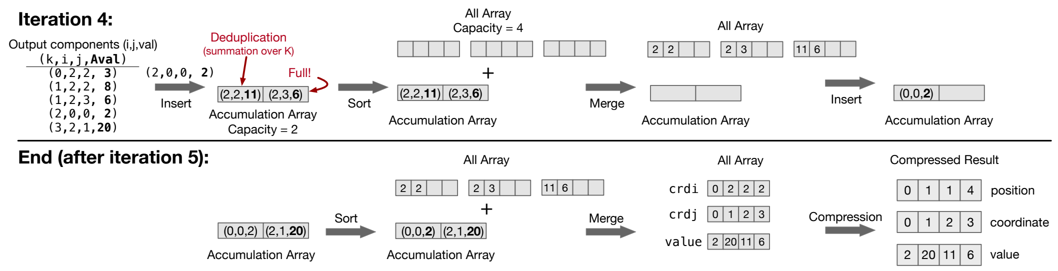


Fig. 7. The insert-sort-merge algorithm template on our outer-product SpGEMM example from Figure 2.

ensure code correctness using a compiler transformation (Section 6.2). Then, we extend the lowerer from CIN to C++ to include the ISM memory pools and function interfaces (Section 6.3).

4 ALGORITHM TEMPLATE FOR SPARSE WORKSPACES

The insert-sort-merge algorithm template (ISM) is a four-stage algorithmic backbone for constructing sparse workspaces. It describes the mechanism we use to insert sparse workspaces into any sparse tensor algebra expression. The compiler emits codes that materialize the template, as described in Section 6.

ISM accumulates the input tensor components, stores them to a temporary array, and finally converts the temporary array to the result's data structure. We leave the concrete memory data structure decisions and function definitions up to the user, allowing for a wide variety of concrete sparse workspace policies as described in Section 5.

4.1 Tensor Component Abstraction

We refer to a nonzero value and its coordinates as a tensor component. The input to the ISM algorithm is a stream of tensor components to be accumulated into the result tensor. As shown in Figure 7, an output tensor component is composed of coordinates `i` and `j`, and the value `val`, which fully describes an element in a matrix. Although tensor components are less memory efficient per nonzero element than other compressed representations such as CSR, they are a direct and clean abstraction that simplifies the algorithm template and code generation.

4.2 The Accumulation Array and All Array

ISM requires storage in which to accumulate components, deduplicate components with the same coordinates, and compress all the generated components to the result. We achieve this by defining two abstract memory pools: the *accumulation array* and the *all array*. The all array is required since a sparse workspace must include at least one memory data structure that stores all of the input components to assemble the output result tensor. To improve performance and manage duplicates, the accumulation array acts as a landing pad that batches the insertions to the all array.

The all array is a temporary linear storage for the result components. Generated components are scattered into it in sorted order, and then the result components are extracted from it to assemble the final tensor format. As shown in Figure 7, the all array stores coordinates in level `I` and `J`, and the values for each unique result tensor component. Finally, the all array is compressed to the result data structure, for example, the CSR format in Figure 7.

We design an accumulation array to serve as an intermediate buffer between the generated tensor components and the all array. The accumulation array can be materialized as any efficient data structure on higher-order tensor components that supports random insertion. In Figure 7, the accumulation array deduplicates at (2,2) by adding the values of components (2,2,3) and (2,2,8). When the accumulation array is full, it is merged with the all array, which we introduce in Section 4.3.

The accumulation array improves performance since it divides and conquers the insertion of components into the all array. We provide evidence of this benefit through a Big-O analysis[1] and empirically in Section 7.5. With the accumulation and all arrays defined, we will describe how they interact in the insert-sort-merge algorithm template to create the result tensor.

Algorithm 1 The ISM algorithm template

```
 1: inputs: Value arrays $val_t$ of tensor $t$, Accumulation
    array $Acc$, All array $All$.
 2: output: Final output tensor $Out$
 3:
 4: $Allocate(Acc)$
 5: while there's still nonzero do
 6:     Iterate and append coordinate to $crds$
 7:     if reach the last level then
 8:         $Insert(crds, Expression(\{Val_t\}), Acc)$
 9:         if $Acc.full$ then
10:             $Sort(Acc)$
11:             $Merge(Acc, All)$
12:             $Insert(crds, Expression(\{Val_t\}), Acc)$
13:         end if
14:     end if
15: end while
16: if not $Acc.empty$ then
17:     $Sort(Acc)$
18:     $Merge(Acc, All)$
19: end if
20: $Compress(All, Out)$
```

4.3 Four-stage Template Model

The input tensor components—generated by the loops that iterate over and compute on sparse and dense tensors—are processed through the insert-sort-merge algorithm and stored into the accumulation and all arrays. Any sparse workspace algorithm must support two types of computation: insertions and deduplications (Figure 1). Insertions place generated tensor components into memory and deduplications sum inserted components that have the same coordinates (collisions). The design of the insert-sort-merge algorithm template, shown in Algorithm 1, distills the sparse workspace construction process into four stages:

(1) **Insertion.** The insertion stage inserts tensor components into the accumulation array.
(2) **Sorting.** The sorting stage triggers when the accumulation array fills up and sorts its components into the order of the result tensor storage.
(3) **Merging.** The merging stage merges the components from the accumulation array into the all array and clears the accumulation array.
(4) **Compression.** The compression stage transforms components stored as coordinates in the all array to the result data format.

In the ISM algorithm, generated components are inserted into the accumulation array during the insertion stage until the array's capacity is reached (Algorithm 1 line 8). Tensor components with the same coordinates are reduced (summed) either during the insertion stage or the sorting stage. When the accumulation array is full, the ISM enters the sorting stage (Algorithm 1 line 9).

Sorting of the accumulation array and merging of the accumulation array into the all array occurs before the accumulation array is cleared and a new tensor component is inserted. The sorting algorithm implementation depends on the choice of accumulation array data structure. We describe multiple concrete sorting and data structure implementations in Section 5 along with their tradeoffs. Next, the merging stage moves components from the accumulation array into the all array, where components remain sorted, and components with equivalent coordinates are reduced. The ISM algorithm separates the sorting phase from the merging phase because sorting can reduce the complexity of merging from $O(n \times m)$ to $O(n+m)$ where n and m denote the number of components in the all and accumulation arrays respectively. When the accumulation array is cleared, the ISM algorithm enters a new iteration.

[1]The detailed analysis can be found in Appendix A in the auxiliary materials [84].

```
#include "ism.h"
// Allocate
AccArray Acc(2,cap,"Coord");
AllArray All(2,cap);
Component c;

// Insert-Sort-Merge
for (int k = 0; k < K; k++) {
  for (int iB = B.pos[k]; iB < B.pos[k+1]; iB++) {
    int i = B.crd[iB];
    c.crd[0] = i;
    for (int jC = C.pos[k]; jC < C.pos[k+1]; jC++) {
      int j = C.crd[jC];
      c.crd[1] = j;
      c.val = B.val[iB] * C.val[jC];
      Insert(c, &Acc);
      if (Acc.full) {
        All.realloc(Acc.size);
        Sort(&Acc);
        Merge(&Acc, &All);
        Acc.refresh();
        Insert(c, &Acc);
      }
    }
  }
}
if (Acc.size > 0) {
  All.realloc(Acc.size);
  Sort(&Acc);
  Merge(&Acc, &All);
}

// Compress
A.crd = All.crd[1];
A.val = All.val;
int* A.pos = (int*)calloc(I+1, sizeof(int));
int iw = 0;
while (iw < All.size) {
  int i = All.crd[0][iw];
  int segend = iw + 1;
  while (segend < All.size &&
         All.crd[0][segend] == i) {
    segend++;
  }
  A.pos[i + 1] = segend - iw;
  iw = segend;
}
int cnt = 0;
for (int pA = 1; pA < I + 1; pA++) {
  cnt += A.pos[pA];
  A.pos[pA] = cnt;
}
```

Fig. 8. Simplified C++ code generated for outer-product SpGEMM following the ISM template. The header file contains definitions for the accumulation array, the all array, and the ISM functions. The compression stage transforms All from COO to the result format CSR.

Table 2. Breakdown of library workspace algorithms into the ISM template for SpGEMM $A_{ij} = \sum_k B_{ik}C_{kj}$.

Library	Insert	Sort	Merge
Gustavson's [28]	1-D coordinate list and flag list along j, deduplication when the flag is true	Bucket sort	Boolean indexing
Cusparse [18]	1-D hash table along j, deduplication when collision	Unsorted	Hash table retrieval
ESC [8]	2-D coordinate list per slice of i	Lexicographic sort	Slice concatenation Reduce-by-key
Buluç's[2] [12]	Heap with key (i, j) per k	Sorted when insert	Multiway merging Reduce-by-key

After the last nonzero element is processed, the remaining components in the accumulation array are sorted and merged into the all array (Algorithm 1 lines 16–18). The last step of the insert-sort-merge algorithm template is to compress the final all array to the expected output tensor format (Algorithm 1 line 20). Figure 8 shows the imperative code that materialize each part of Algorithm 1 for outer-product SpGEMM example in Section 2.1.

4.4 Recreating Prior Work with the ISM Framework

The insert-sort-merge algorithm template provides a general and modular framework to assemble various concrete implementations. We will show how existing workspace algorithms from prior work are expressed in terms of the ISM template. As shown in Table 2, prevailing hand-optimized workspace algorithms for SpGEMM can be expressed in ISM by implementing different sorting and merging algorithms. Since these algorithms do not use an accumulation array, the merge stage column in Table 2 shows how their workspaces are compressed to the output.

[2]We list the core ideas of Algorithm I of Buluç's paper, but the actual algorithm computes the multiway merging on the fly.

For example, Gustavson's algorithm uses a flag array to label positions that already have nonzero elements. Therefore, it can deduplicate components upon insertion by checking the flags. Meanwhile, the components are sorted because the position in the 1-D coordinate list is equal to the level J index, which is based on a bucket sort [20]. The positions where the workspace flag is set are merged to the output; this procedure is also known as boolean indexing [16]. Other algorithms can be analyzed similarly.

5 CONCRETE SPARSE WORKSPACE POLICIES

As demonstrated in Section 4.4, there are many different concrete options for each stage of the ISM template. A user can synthesize many different concrete workspace policies (or algorithms) by mixing different data structure, sorting algorithm, and optimization implementations. This section introduces some of these implementation decisions we made to demonstrate a few new techniques that are compatible with our template. The concrete sparse workspace implementations introduced in this section also give a flavor of the types of workspace optimizations possible with our approach and allow us to evaluate our abstract template concretely in Section 7.

5.1 Data Structures

Our compiler materializes the all array in a COO data structure. We chose this data structure because COO is convenient for transforming the physical organization of sparse tensor components [27, 51, 72], and we leverage work on code generation for format conversions between COO and other formats from the literature [15].

The accumulation array serves as a buffer between the input components and the all array. Sparse iteration scatters components into the accumulation array, and then those componets are sorted and merged with the all array. Therefore, the accumulation array data structure should support efficient deduplication, sorting, and sequential accesses.

Accumulation Indexing. A strategy that minimizes data movement is to store the tensor components in an array, where the components are not moved until the accumulation array is freed. Sorting and merging are executed on the indices of each component in the array rather than on the component structs themselves. In this way, only integer indices are moved, reducing memory footprint to about $\frac{1}{M+1}$ where M denotes the mode of input components. During merging, the sorted accumulation indices are used to access the component with the smallest coordinates in the accumulation array. An array structure, however, is not always the best choice; other valid data structures supported by the ISM template may not support this indexing optimization.

Reallocation. We can resize the accumulation array after it is merged in order to avoid frequent merging with the all array. We use a three-stage piecewise linear function with heuristic thresholds and slopes to determine our allocation size. This heuristic allocation policy avoids memory overallocation when compared to a naive memory reallocation policy, like memory doubling.

5.2 Sorting Algorithms

When the accumulation array reaches its capacity, the indices are sorted based on the coordinates of the tensor components. This computation can be modeled as the sorting of multiple arrays where each corresponds to one level of the result tensor. Although there are various multi-array sorting algorithms (e.g., quick sort, bucket sort, and counting sort [11, 45]), we only implement two sorting algorithms that leverage the unique traits of the ISM sorting stage. In the ISM arrays, each component has a unique label, represented by a finite positive integer and a known range. For example, for a matrix with shape (I, J), the component with coordinates (i, j) can be labeled using

$(i \times J + j)$, and the coordinates of the first mode are in the range $[0, I)$. The two sorting algorithms may be combined with each other because they are each defined to sort a single tensor level.

Bucket Sort. This algorithm sorts the indices of the accumulation array (accumulation id) by a bucket B parametrized by L and h. Each bucket holds a list of accumulation indices. L is the length of the bucket, and h is a function that maps the coordinates within a component to an integer (the bucket id) in range $[0, L)$. Example mapping functions for the component with coordinates (i, j) of an $I \times J$ matrix include $h(i, j; I, J) = i$ when $L = I$ and $h(i, j; I, J) = (i \times I + j)\%L$ when $L < I$. The bucket sorting algorithm also includes a boolean flag array that records whether an element has been inserted. If B[bucket id] is empty, the algorithm will allocate a list at B[bucket id] and insert the accumulation id of the inserted component. If B[bucket id] is not empty and there is already an accumulation id in B[bucket id] whose component has the same coordinates as the one being inserted, the values of the two components are summed. Otherwise, the accumulation id is appended to B[bucket id].

Coordinate Sort. This algorithm appends inserted accumulation ids to a list, whose length is the capacity of the accumulation array. Unlike the bucket sort algorithm, this sorting algorithm does not reduce values upon insertion. When called by the ISM template, the coordinate sort applies a 1-D array sort on this list with the comparison function defined as the lexicographic order of each level of coordinates. Like bucket sort, the coordinate sort returns a sorted list of accumulation ids.

5.3 Optimizations

We utilize the staging of the ISM structure to accelerate computation. The insertion stage only writes to the accumulation array at the current time step, which is independent of the sort and merge stages of the previous time step. Therefore, we utilize such independence by pipelining them using multiple threads. Moreover, the all array at the next time step is independent from that of the previous time step. Therefore, we can double buffer the all arrays to avoid auxiliary array creation during merging. Both optimizations are orthogonal to the data structure and sorting algorithm decisions described previously.

Pipelining (Stage Level Parallelism). The sort and merge stages can be pipelined with the insertion stage. When the accumulation array is full, our implementation spawns another thread to execute the sort and merge, and the main thread continues to execute the insert for the next sparse iteration step. In this way, the latency of the sort and merge is hidden. However, we now need to allocate two accumulation arrays and switch them between threads when one of them reaches capacity. Figure 9 shows this—the first accumulation array becomes full at t_1, and the second accumulation array is inserted into between t_2 and t_1 while the first one is being merged concurrently.

Double Buffering. Having two copies of the all array can avoid temporary array allocations. We implement the merge step as an out-of-place merging of two arrays. If there is only one all array, every merge must allocate a temporary array to hold the merged results before dumping its values into the all array. If we double buffer the all array, then we can eliminate this temporary array. Figure 10 compares these two merging approaches.

In the ideal case, the execution time of any workspace policy (and the ISM template) is negligible compared to the total kernel execution time. We provide an ablation study for various policies and compare the empirical results with the lower bound runtime in Section 7.3.

5.4 Towards Parallel Sparse Workspaces

Although our evaluation is limited to sequential code with sparse workspaces, the insert-sort-merge template is not tied to sequential execution. In general, ISM enables a decoupled access-compute

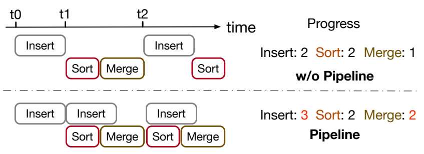


Fig. 9. Stage level pipelining of the sort and merge stages with insert.

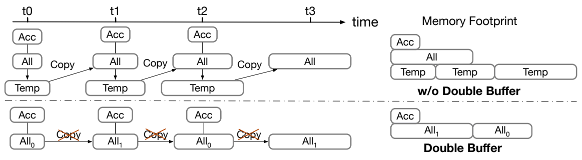


Fig. 10. Double buffering the all array, which produces a lower memory footprint.

parallel pattern [68] where parallel producers perform sparse iteration and insert components into the accumulation array, while parallel consumers merge the accumulation array with the all array. With proper synchronization strategies between parallel producers and consumers, our framework can be extended to support parallel sparse workspace. The performance challenges lie in how to increase data locality [25], reduce atomic operations [54], and balance workloads [3]. Prior kernel-specific work has proposed solutions for these challenges for hand-written kernels [19, 46, 50, 60]. The accumulation array can be tiled to better utilize specific architectures [56]. The all array can be avoided by precomputing the output matrix structure in an extra symbolic phase [59]. ISM provides an abstract analysis of these kernels and serves as a foundation for developing code generators that can generate parallel sparse workspaces. We leave exploring the tradeoffs of parallelism-enabled sparse workspace code as future work.

6 COMPILATION

This section describes the new compiler techniques of our system. We first introduce new scheduling and format language commands such that users can manually express tensor expressions with sparse workspaces in our compiler (Section 6.1). To increase productivity, our compiler also has the ability to automatically detect sparse scattering behavior within expressions and then automatically transform those expressions to include missing sparse workspaces for correctness (Section 6.2). Once our compiler detects that a CIN expression contains a sparse workspace tensor, it automatically generates code that inserts the insert-sort-merge algorithm template as described in Section 4.

6.1 Sparse Workspace Scheduling and Format Commands

Our compiler extends the TACO scheduling language to allow the insertion of sparse workspaces into index notation expressions. We also extend the where statement in CIN, which precomputes an expression into a temporary tensor variable [38], to describe parameters that configure the ISM algorithm for code generation and attributes of the sparse workspace arrays. Users can either directly use the scheduling language to insert sparse workspaces as shown in Figure 12, or rely on our automatic sparse workspace insertion algorithm as shown in Figure 13.

Users invoke the sparse workspace transformation via the `precompute` scheduling command [38], whose C++ declaration is as follows:

```
void IndexStmt::precompute(IndexExpr expr, vector<IndexVar> i_vars,
                                        vector<IndexVar> o_vars, TensorVar ws);
```

This command transforms a CIN statement that contains a sub-expression `expr` to a statement with a where sub-statement $\ldots = \ldots \texttt{ws}_{\texttt{o\_vars}}$ **where** $\texttt{ws}_{\texttt{i\_vars}} = \texttt{expr}$. We call the left-hand side of the where statement the consumer as it consumes the workspace and the right-hand side of the where statement the producer as it produces the workspace data. The producer is a transformed CIN statement with the result tensor of the sub-expression `expr` replaced by the `ws`. The consumer assigns the result tensor in the original CIN statement to a transformed expression that uses the `ws`. The command also optionally replaces the index variables from the original statement `i_vars` on the consumer side with the corresponding `o_vars`. For simplicity, we assume `o_vars` is equivalent to `i_vars` for the rest of Section 6. Our compiler's `precompute` command transformation differs in the tensor variable (`TensorVar`) type for the workspace and in the construction of the consumer.

Our additions to the tensor variable format include a sparse format `SpFormat` class, the output order of the workspace `ow_order`, and additional metadata to the `TensorVar` class. The SpFormat class configures the materialization of the accumulation array. It annotates the specific sorting algorithm by an enum in the SpFormat class (signified by the `Coord` argument in Figure 12), which also assigns the materialized data structure for the accumulation array. The SpFormat class also annotates the number of orders of the accumulation array, which equals the length of the `ow_order`. The `ow_order` assigns how the workspace's access order is converted from the producer to the consumer. A concrete algorithm that automatically decides the appropriate `ow_order` is described in Section 6.2. The TensorVar metadata stores the dimensions of each level in the all array and the parameters required for the chosen sorting algorithm. For example, the coordinate sort (`Coord`) algorithm requires the initial capacity of the accumulation array.

These scheduling and format commands expand TACO's scheduling space by introducing additional scheduling options to configure workspaces. They thus alter the performance model of different schedules. Sparse workspaces remove key constraints on scheduled expressions by legalizing sparse scattering behavior. For the performance objectives, sparse workspaces add an intermediate stage to the computation flow, which may favor different types of data locality caused by the user-provided schedules.

6.2 Automatic Sparse Workspace Insertion

The automatic sparse workspace insertion transformation decides whether or not a sparse workspace should be inserted and, if so, which sparse workspace configurations to use. Users can apply schedules like `split`, `pos`, and `reorder` to a CIN expression as if no sparse scattering behavior exists in the expression. Then, the compiler automatically detects whether sparse scattering occurs and, if necessary, inserts a workspace into the CIN.

The algorithm deduces the loop order `input_order` from the CIN expression and compares it with the access order of the result tensor variable `output_order`. If these two orders are the same, then the expression is *concordant* [1]. Otherwise, it is *discordant*. If every level of the result tensor supports random insertion and lookups (i.e., it behaves as a dense level format), then we do not insert any workspaces. Even if the expression is discordant, the order mismatch may not require a workspace because it only involves linear transformations on the access order, like $i \times J + j \longrightarrow j \times I + i$, which is expressible using dense formats. Otherwise, if the expression is discordant, we assign the dimension of each level of the SpFormat to be the same as the output, and `ow_order` is calculated to satisfy the following constraint `input_order[i] == output_order[ow_order[i]]`. If the expression is concordant, we check the mode formats of all tensors. If the result format has different storage levels than the iteration levels, we insert a sparse workspace and transform the consumer-side assignment to the format conversion IR of Chou et al. [15]. If the result tensor format's storage levels are the same as its iteration levels, we can either insert a workspace with the same levels and dimensions as the result or use the common iteration hoisting optimization (see Section 6.2) to allocate a lower-order sparse workspace.

Input Order Reconstruction. First, the transformation must deduce the loop order using the original (unscheduled) index variables. The key step in this deduction is to reconstruct the original loop order from the expression by remapping transformed index variables back to their original indices. This retrieval may be complex since the expression may already be composed of other schedules that transform and change the loop index variables without modifying the access index variables [65]. The transformation collects the input-loop index variables `input_order` from the

```
A: ({Compressed,Compressed}, {j,i})→{j,i};
B: ({Dense,Compressed}, {i,k})→{i,k};
C: ({Dense,Compressed},{k,j})→{k,j};
stmt: A(j,i) = B(i,k) * C(k,j);
stmt = stmt.reorder({i,k,j})
           .fuse(i,k,f)
           .pos(f,fpos,B(i,k))
           .split(fpos,{f0,f1},4)
           .reorder({f0,f1,j});
```

Fig. 11. Example input schedule to the Automatic Sparse Workspace Insertion: $\forall_{f_0 f_1 j} A_{ji} \mathrel{+}= B_{ik} C_{kj}$ s.t. $fuse(i, k, f)$ and $pos(f, f_{pos}, B_{ik})$ and $split(f_{pos}, \{f_0, f_1\}, 4)$

```
W: Tensor(SpFormat(2, Coord), {I,J}, {1,0}, cap);
stmt = stmt.precompute(B(i,k)*C(k,j), {i,j}, {i,j}, W);
```

Fig. 12. The code required to manually insert a workspace for the example in Figure 11. The updated CIN now contains a where node: $\forall_{ji} A_{ji} = W_{ji}$ where $\forall_{f_0 f_1 j} W_{ij} \mathrel{+}= B_{ik} B_{kj} \ldots$

```
stmt = insertSparseWorkspace(stmt, Coord, cap);
```

Fig. 13. The automatic sparse workspace insertion command is equivalent to Figure 12.

composition of ∀s in the input expression. Then, our compiler has three rules to retrieve the access index variables from the loop index variables for `fuse`, `pos`, and `split`, respectively. These rules are applied in the inverse order of the scheduling relations in the expression. Our compiler targets the following three scheduling commands [65]:

- **split**. `split(IndexVar i, IndexVar i0, IndexVar i1, size_t s)` splits an index variable i into two nested index variables (`i0` and `i1`) with the iteration step `s`.
- **fuse**. `fuse(IndexVar i, IndexVar j, IndexVar f)` fuses two nested index variables (`i` and `j`) to an index variable `f`.
- **pos**. `pos(IndexVar i, IndexVar p, Access A)` replaces the coordinate space index `i` with a transformed index variable `p` that iterates through the position space of input access `A` over the same iteration range.

If a `split` relation occurs, the transformation detects the locations of the split index variables, `i0` and `i1`, in the loop order and determines which is the inner index and which is the outer index position. Then, if `i` is a reduction variable, the transformation replaces the outer split index variable with `i` and deletes the inner split index variable. Otherwise, it replaces the inner one with `i` and deletes the outer. This distinction is designed for optimizing common non-reduction iterations. The split transformation does not change the start position of the reduction so the outermost index (between `i0` and `i1`) decides the insertion position for `i`. If a `pos` relation occurs, the transformation replaces the index variable `p` with `i` since `pos` does not change the access order. If a `fuse` relation occurs, the transformation replaces the index variable `f` with `i` and `j` since `i` and `j` were consecutive in loop order when they were fused.

For example, consider an SpGEMM followed by a matrix transposition $A_{ji} = \sum_k B_{ik} C_{kj}$. We assume the user imposes the following schedules on the expression, as shown in Figure 11. The initial value of `input_order` equals $\{f0, f1, j\}$. Based on the above rules, we construct the `input_order` as $\{f_0, f_1, j\} \Rightarrow \{f_{pos}, j\} \Rightarrow \{f, j\} \Rightarrow \{i, k, j\}$.

Input-output Comparison. This step of the transformation compares the input loop order with the access index variables of the result tensor. First, the transformation identifies the access order of the result tensor `output_order`. Then, the transformation eliminates any variables in the `input_order` that are not used in the result access. The size of the final `input_order` set determines the number of workspace levels and the final `input_order` can now be used to compare with `output_order` to deduce the correct `ow_order`.

For the SpGEMM example, the result tensor access A_{ji} defines `output_order` as $\{j, i\}$. The order does not contain a k index so k is a reduction variable. This step eliminates k from the `input_order` $= \{i, k, j\}$ to produce the final `input_order` $= \{i, j\}$. The sparse workspace must have two levels with dimensions $\{I, J\}$, which are assembled in the `input_order` $\{i, j\}$ and accessed in the `output_order` $\{j, i\}$. The `ow_order` must be $[1, 0]$ to satisfy the constraints `input_order[0] == output_order[1]` and `input_order[1] == output_order[0]`. The final generated workspace W for this example is shown in Figure 12 on line 10.

Common Iteration Hoisting. If part of the `input_order` and `output_order` match, common iteration variables can be hoisted out of the where statement to become nested foralls that serve as shared outer loops for both the consumer and producer. The conditions for this optimization are:

(1) The output tensor format has the same storage levels as the iteration levels.
(2) The index variables are the same in both orders until one position $p > 0$ where the index variables mismatch. All index variables after position p are ignored.
(3) For all the index variables that match before position p, the corresponding modes in the output tensor have stronger abilities than the input tensors.

We measure a tensor's ability using the same method as Ahrens et al. [1]. A tensor's ability is the combination of the abilities of its level formats. Intuitively, a format's ability measures the time complexity of inserting and accessing the format. If the result format can be assembled using the access pattern generated by the input tensors, we define the result format as stronger than the input. In other words, if the result format is stronger than the input, the result can be directly assembled without the help of a workspace. For example, a dense output is stronger than a compressed input because the dense format supports inserts whereas the compressed format needs to be iterated by position and writes to the result via insertion.

Dense Workspace Optimization. This optimization is an extreme case of common iteration hoisting, but we define it as a separate case since it allows our compiler to support dense workspaces. If an expression is concordant, it may be identified as having dense scattering behavior, which can be solved by a dense workspace. Dense workspaces may perform better than sparse workspaces in this scenario because the sparse workspace has to store extra index arrays that are not needed by the dense workspace. The conditions for this optimization are:

(1) There is a reduction variable in the `input_order` retrieved from the schedules. In our SpGEMM example, $\{i, k, j\}$ contains the reduction variable k.
(2) There is only one variable in the `input_order` after the reduction variable. Our outer-product SpGEMM example does not satisfy this constraint.

If these requirements are met, our transformation uses the dense workspace transformation of Kjolstad et al. [38]. We analyze the performance of the dense workspace optimization in Section 7.2.

6.3 Code Generation

After the sparse workspace configurations are set by the scheduling language, our compiler lowers CIN to a C-like imperative IR before finally generating C++ code. The C-like imperative IR models platform-agnostic basic blocks such as variable declarations, loops, conditional statements, and function calls. Our imperative IR introduces a new mechanism for the accumulation and all arrays and generates ISM as function calls to external functions. During code generation, our compiler integrates ISM function calls and variables into the generated code, which materializes the specific ISM policy assigned by the user in the scheduling language.

Our code generation algorithm for sparse workspaces occurs whenever the compiler detects a where statement with a SpFormat tensor variable. The sparse workspace tensor variable is accessed in two different ways on the producer and consumer sides, while an ordinary tensor is only accessible in one way. Our compiler uses the same sparse iteration generation mechanism as TACO, transforming tensor access index variables into per-level iterators [39]. The level format of the input tensors determines the attributes and capabilities of the loop iterator. The workspace has the same access attributes as a dense level: random insertion and lookups. Therefore, when generating producer-side co-iteration code, our code generator treats the sparse workspace as a dense tensor and replaces the assignment statement with the ISM functions. The rest of the code generation algorithm follows from the TACO body of work [15, 65]. Our compiler modifies code generation in the following steps:

(1) Our compiler inserts the proper data structure allocations when an SpFormat is detected in the CIN expression. See lines 4–6 in Figure 8 for an example of the generated code.
(2) To force concordance, our compiler tracks the tensor component's coordinates and reorders them based on the `ow_order`. This behavior generates a variable assignment expression in the body of any forall loop that has an index variable participating in the `output_order` of the result tensor. See lines 12 and 15 in Figure 8.
(3) When our compiler detects a where statement with an associated SpFormat workspace tensor, the mechanism forces any loop index variable iterators that are also in the sparse workspace producer access to have dense capabilities.
(4) Then, our compiler detects the sparse workspace assignment on the producer-side of the where statement and inserts the ISM function calls. See lines 17–23 in Figure 8.
(5) Our compiler emits the cleanup ISM function calls between the producer and consumer sides of the detected where statement. See lines 28–32 in Figure 8.
(6) Finally, our compiler lowers the consumer side, which emits the compression function. When generating the compression function, the relevant consumer-side access iterators are created using (higher-order) COO format capabilities. See lines 34–53 in Figure 8.

The code generation algorithm allows our compiler to fill in the code holes produced by the ISM template using user-defined functions. We purposefully separate the code generation of the ISM function template from the IR for modularity. Our template design allows for different optimizations and policies, and we anticipate that users will want to introduce other hand-optimized implementations in the future. Our system is robust to such changes and sufficiently modular to isolate the optimizations to the ISM function definitions. However, the ISM template is encoded directly into the IR transformations of our system since we have shown the generality of the ISM template structure in expressing various policies.

7 EVALUATION

We evaluate our sparse workspace technique by comparing the performance of linear and tensor algebra kernels against state-of-the-art systems. We also perform ablation studies on different sparse workspace policies to describe the optimization space of our design.

7.1 Methodology

All experiments are run on a dual-socket 2.4 GHz Intel Xeon E5-2640v4 machine with 40 cores (80 threads) and 50 MB of L3 cache per socket, running Ubuntu 20.04 LTS. The machine has 256 GB of memory and runs kernel version 5.4.0-155-generic and GCC 9.4.0. We compare our work against TACO at commit 2b8ece4 [38], Eigen version 3.4.0 [26], and Cyclops Tensor Framework (CTF) at commit e52330f [67]. We implement our compiler in C++ as an extension to the TACO compiler. All experiments are run with 5 warmup rounds and report the arithmetic mean execution time of 20 benchmark rounds. As in prior work [31], all SpGEMM kernels are run with the compressed (sparse) matrix operand B and the second operand is that same matrix transposed B^T with the columns shifted by one. We report all runtimes and memory usage in log-scale.

Figure 14, Figure 17, and Figure 18 were run on real-world data from the SuiteSparse matrix collection [42]. To select a representative and diverse set of matrices, we randomly sampled 20

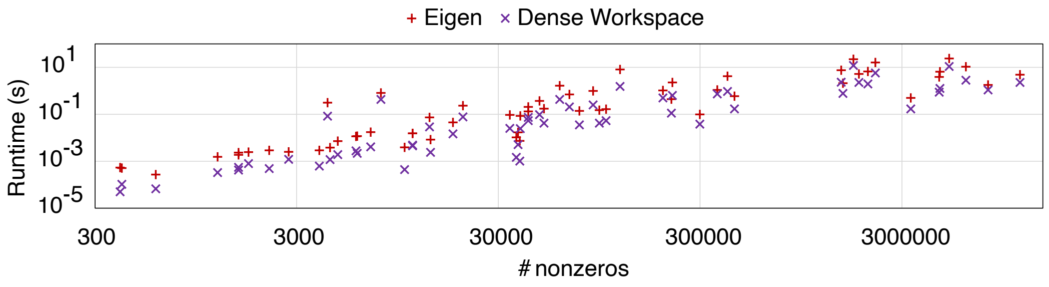


Fig. 14. TACO with dense workspace vs. Eigen on row-wise SpGEMM for selected SuiteSparse matrices.

matrices from each category based on the number of rows: less than 12000, 12000 to 180000, and larger than 180000. Tensors in this section that only have their density or mode defined are sparse based on a uniform random distribution.

In this section, a sparse workspace with a "Bucket" policy denotes a bucket sort with $L = I$ and $h(i, j) = i$, a "Hash" policy denotes a bucket sort with $L < I$ and $h(i, j) = (i * J + j)\%L$, and a "Coord" policy denotes the coordinate sort. For the "Hash" and "Coord", we heuristically assign L as $2^{\lceil log_2 nnz \rceil}$, where nnz is the number of nonzeros of the input matrix.

7.2 First-order Sparse Workspaces in Linear Algebra

Our compiler can generate dense workspace code competitive with a hand-optimized library for first-order (vector) workspaces as expected. In this case, our sparse workspaces are efficiently implemented using the dense workspace technique from prior compilation systems. Specifically, in Figure 14, we compare against Eigen, a state-of-the-art linear algebra library, for the row-wise SpGEMM with all matrices stored in CSR format.

Though the generated dense workspace kernel outperforms the Eigen library, there exist cases where first-order sparse workspaces, such as those generated by our compiler, outperform both Eigen and dense workspaces. We evaluate the sparse workspace using the same SpGEMM expression with CSR matrices but on synthetic data. In these experiments, we pick the Bucket sparse workspace policy for all linear algebra expressions. The input matrix B is generated with 10% nonzero elements along level K, and the dimension of level K is swept from 2500 to 40000 with increments generated on a logarithmic base 2 scale. Each column has 1000 nonzeros, and we keep the number of dense elements as 10000×10000. Our synthetic data lets us indirectly control the number of collisions that occur during accumulation, as the dimension of level K increases the amount of coordinate deduplication also increases. As shown in Figure 15, the sparse workspace is better than the dense when there is less deduplication. Moreover, first-order dense workspace memory grows proportional to the dimensions of one level. When the amount of deduplication is small, the sparse workspace still consumes less memory because the output tensor components are sparse. However, as the deduplication increases, the dense workspace improves due to better locality.

7.3 Second-order Sparse Workspaces in Linear Algebra

Sparse workspaces use less memory than dense workspaces, which is important when a temporary tensor is very sparse, as a dense workspace may not fit in memory. Moreover, sparse workspaces may improve performance over dense workspaces, due to increased memory locality.

We show the performance and memory benefits of sparse workspaces over the dense baseline on second-order scattering in the outer-product SpGEMM expression. For the compiler-generated algorithms, the input matrices are stored in doubly-compressed sparse row and column (DCSR and DCSC) formats [12] since it compresss out the columns without deduplication. For Eigen, the input

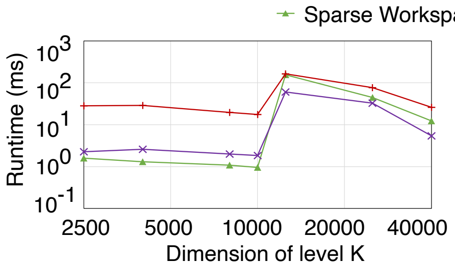


Fig. 15. Comparison of sparse and dense workspaces on row-wise SpGEMM for synthesized matrices.

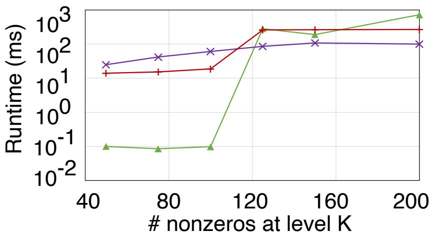


Fig. 16. Comparison of sparse and dense workspaces on outer-product SpGEMM for synthesized matrices.

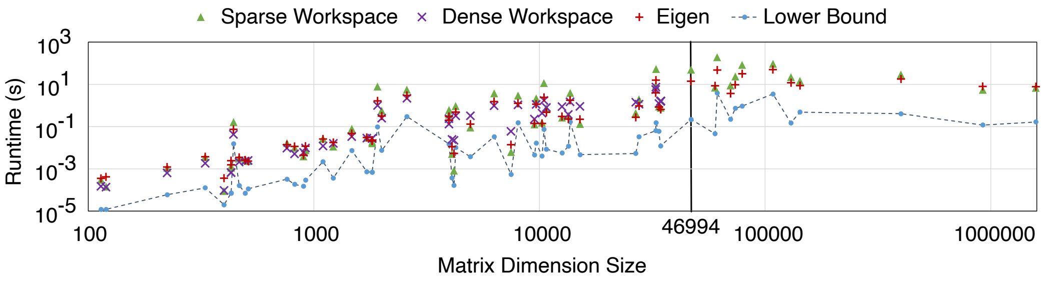


Fig. 17. Comparison among the best results of different sorting algorithms with the dense workspace on second-order scattering. We construct the lower bound as described in Section 5.

matrices are stored in CSR since it does not support DCSR. The result matrix is stored in CSR so both methods need to compress the workspace to the output format.

The trend in Figure 16 is similar to that of Figure 15. Again, the sparse workspace is better when there is less deduplication, but the second-order workspace has greater overall speedup since higher orders have a larger opportunity for savings. Unlike Figure 15, we keep the dimensions of I and K as 10000 and 10000, respectively, for Figure 16 and sweep the number of nonzeros at level K. We still keep 1000 elements in each column with nonzeros in matrix B.

As shown in Figure 17, the performance of sparse workspaces remains competitive even where a dense workspace fits in memory. Also, the sparse workspace only incurs overhead proportional to the number of sparse iterations because the sparse workspace scales at the same rate as the runtime lower bound. Following the definition of the runtime lower-bound in Section 5.3, we record the ideal execution time by only keeping the sparse iteration computation and eliminating all the code related to the workspace.

A sparse workspace is necessary for sparse scattering when a dense workspace is too large to fit in memory. As shown in Figure 18, the sparse workspace also saves on average 3.6× the amount of memory for the matrices that do not OOM. We estimate the memory footprint as $shape \times (3 \times 4 + 1)$ bytes for the dense workspace and $nnz \times 3 \times 4$ bytes for the sparse workspace, where $shape$ is the number of dense elements, and nnz is the number of nonzeros. We confirm this trend in Figure 18, which demonstrates that the dense workspace runs out of memory when the input matrices get too large (after the black line), while the sparse workspace scales.

7.4 Sparse Workspaces in Tensor Algebra

Sparse workspaces are especially important for higher-order tensor algebra expressions, as they tend to need higher-dimensional temporaries. We demonstrate this effect for SpMTTKRP $\forall_{klij} A_{ij} = B_{kli} C_{kj} D_{lj}$, SpTTM $\forall_{kijl} A_{ijl} = B_{kij} C_{kl}$ and SpTTM-I $\forall_{kijl} A_{kjl} = B_{kij} C_{il}$, which are used to factorize

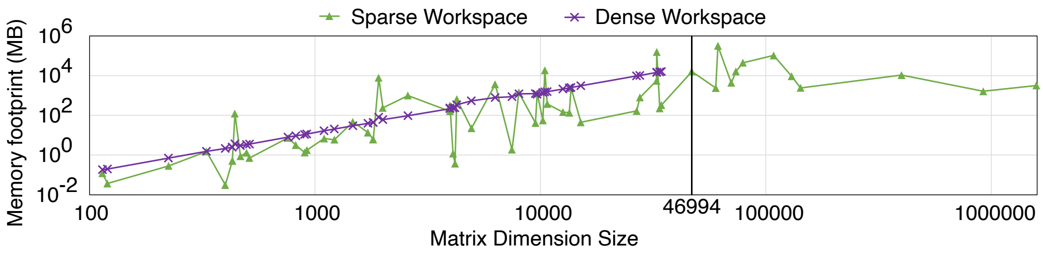
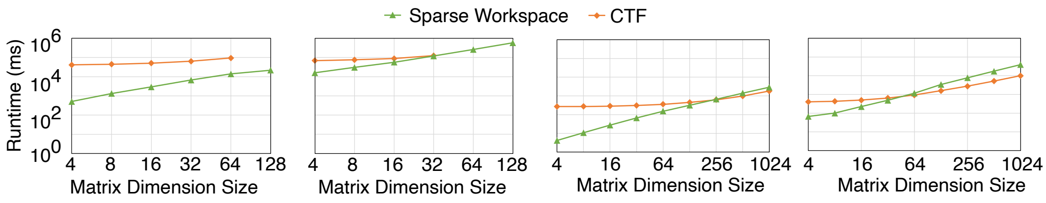


Fig. 18. The memory footprint of sparse vs. dense workspaces on second-order scattering.



(a) SpMTTKRP on nell-2. (b) SpTTM on nell-2. (c) SpMTTKRP on uber3. (d) SpTTM on uber3.

Fig. 19. SpMTTKRP and SpTTM runtime on nell-2 and uber3.

tensors [47, 69]. We test these expressions on the nell-2, uber3, and nips3[3] from the FROSTT dataset [70], and the Facebook tensor [76] that fit in our memory.

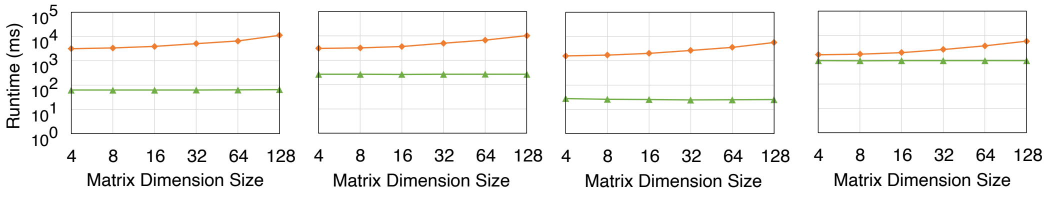
As shown in Figure 19, we compare the generated Hash sparse workspace to CTF, a state-of-the-art sparse tensor algebra library [67]. We sweep the dimensions of the free level of the matrix (J for SpMTTKRP and L for SpTTM) and keep the density of the input matrices as 10%. For both systems, we store input higher-order tensors in the compressed sparse fiber (CSF) [71] format and output tensors as COO. The input matrices to CTF are stored in COO since that is the only format it supports for matrices. CTF reduces higher-order tensor contractions to matrix multiplications through index folding, which costs time and memory. Therefore, CTF runs out of memory in Figure 19b and Figure 19a. Also, CTF computes some metadata of the tensors for optimization before computation. Such overhead is apparent when the matrix is small. When the matrix grows larger, the benefits from the metadata outweigh the latency of the preprocessing. Therefore, in Figure 19c and Figure 19d the runtime of the sparse workspace grows faster than CTF as the matrix dimension increases. We also evaluate SpMTTKRP on the freebase_sampled tensor from Freebase [33] with J = 4. CTF OOMs in this case, while our method takes around 16 hours to finish.

To show that the performance of sparse workspaces is agnostic to the shape of the input tensors, we do experiments in Figure 20. Unlike Figure 19, we keep the input non-zero elements unchanged and sweep the dimensions of the free level of the matrix. The other experimental settings are the same as Figure 19. As expected, the runtime of the sparse workspace does not increase as the matrix dimension size becomes larger since the sparse iteration and ISM behaviors do not change. On the contrary, the runtime of CTF increases because it uses dense sub-tensors to support its index folding, whose cost grows with the size of the matrix.

7.5 Study of Sparse Workspace Design Choices

We perform an ablation study to analyze the different optimizations and concrete policies of our sparse workspace generation framework. We first empirically justify the two-level accumulation

[3]We modify the FROSTT uber and nips 4-tensors to 3-tensors by dropping one dimension as in Hellsten et al. [29].



(a) SpMTTKRP on fb. (b) SpTTM-I on fb. (c) SpMTTKRP on nips3. (d) SpTTM-I on nips3.

Fig. 20. SpMTTKRP and SpTTM-I runtime on Facebook (fb) and nips3.

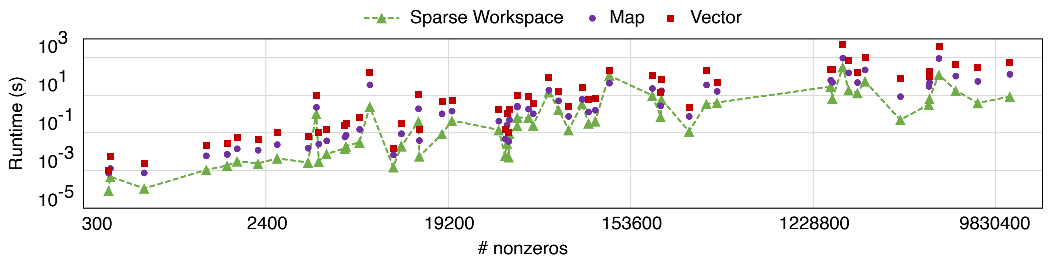


Fig. 21. Empirical benefits of the accumulation and all array structure in our ISM template. We compare our bucket policy against a map and vector data structure policy. The map implicitly sorts every insert (accumulation size = 1), whereas the vector only sorts once before compressing the output (accumulation size = sizeof(output)), which are the two extremes of our ISM template.

and all arrays, instead of only using one array as in prior work policies [44, 66]. We then investigate the influence of the concrete policies introduced in Section 5.

In Section 4.2, we analyzed the benefit of using a two-level array structure via Big-O analysis. However, since Big-O can be misleading in practice, we compare our optimized methods with some straightforward sparse workspace policies that only use one storage array. Our two-array workspace policies have two extremes. One extreme sorts and deduplicates every time an output component is inserted, which occurs when using a single map as a sparse workspace [66]. The other extreme is to sort and deduplicate after all output components are generated, which occurs when inserting components into a single vector with one sorting and deduplication pass at the end before compression. In both of these extremes, only a single data structure is necessary. Our proposed method lies in the middle because we sort and deduplicate in batches using the accumulation array and merge the components with the all array every time the accumulation array is full. Though the two extremes do not need a merging stage, all three methods require a compression stage. Figure 21 shows that these two extremes—the map and the vector—are slower than our bucket policy in most cases. Therefore, it is empirically beneficial to use our ISM algorithm template with two arrays.

In Section 5.2 and Section 5.3, we introduced two concrete sorting algorithms and two optimizations for sparse workspaces that fit within our ISM framework. Figure 22, Figure 23, and Figure 24 show that no single sorting algorithm or optimization can dominate all inputs. As shown in Table 3, this observation still holds for higher-order tensors. In Table 3, we sweep the dimensions of the free level J for SpMTTKRP and keep the density of the input matrices as 1%[4]. Since the performance of sparse computation is data-dependent, the compiler should support many concrete sparse workspace policies to maintain performance across the range of sparse data.

[4]Other results can be found in Appendix B, and they share similar trends. The full lists of matrix and tensor data we use can be found in Appendix C and D in the auxiliary materials [84].

Table 3. SpMTTKRP runtime (ms) of sorting algorithms on nips3. We underscore the best policy for each J.

Method	J=4	J=8	J=16	J=32	J=64	J=128	J=256	J=512	J=1024	J=2048
Bucket	0.4390	1.037	3.022	7.904	27.41	72.40	165.3	322.1	664.6	1331
Hash	0.5954	1.204	3.139	7.576	26.08	64.65	152.3	317.9	673.5	1410
Coord	0.3422	0.9175	2.883	7.687	27.18	71.5	163.0	336.7	706.7	1388

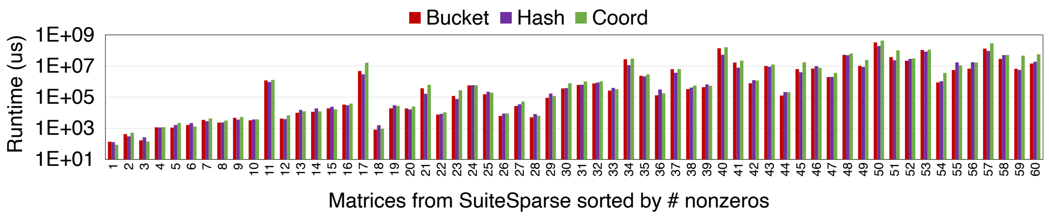


Fig. 22. Runtime comparison among different sorting algorithms.

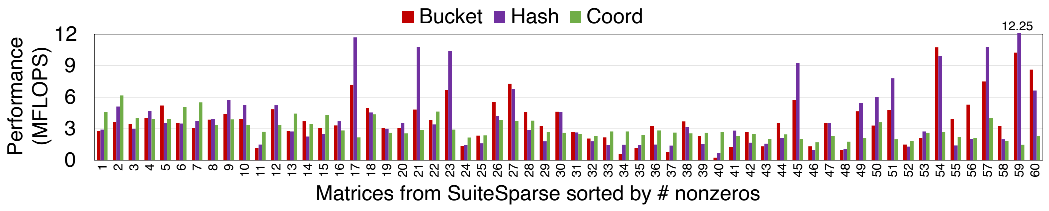
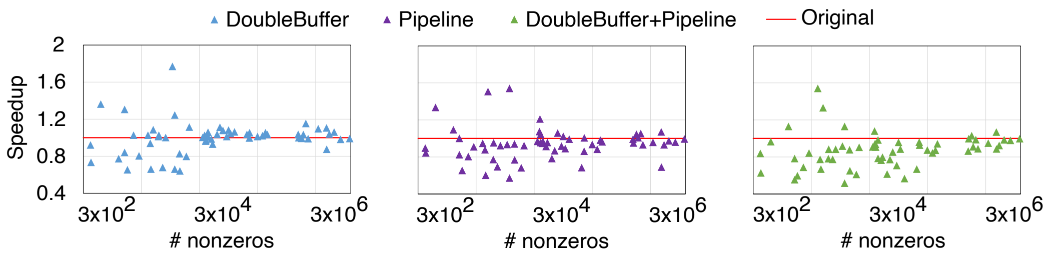


Fig. 23. Performance comparison among different sorting algorithms reported in a linear scale. FLOPS is calculated as the number of fused multiply-add (`fma`) operations divided by the runtime.



(a) Double Buffer. (b) Pipeline. (c) Double Buffer + Pipeline.

Fig. 24. Bucket sort using two optimizations proposed in Section 5.3. Speedups are reported in a linear scale.

8 RELATED WORK

We introduce four areas of related work. The first area, code composition in compilers, describes our work in terms of general compilation strategies from prior work. Then, we focus specifically on works related to workspaces and sparse systems, namely workspace optimizations in sparse kernels, workspace optimizations in sparse compilers, and sparse tensor format abstraction.

Code Composition in Compilers. Code composition enables the modular reuse of high-performance codelets (or stencils) without modifying the compiler cross-layer. Exocompilation [32] provides a framework that replaces computation procedures with hardware primitive functions. Mosaic and TVM [7, 13] integrate vendor library functions into their DSL lowering process. Template JITs [21, 22, 61, 80] apply a similar methodology to general compilation. However, they integrate pre-defined bytecodes or AST nodes instead of library functions. Our method is most similar to Hyper [55], FFTW [23], and the ideas in [73] in that the compiler composes user-defined algorithmic templates with the rest of the generate code.

Workspace Optimizations in Sparse Kernels. Prior work on individual tensor algebra kernels has used workspaces for both dense and sparse scattering. The dense workspace was first used to implement the multiple-switch algorithm in SpGEMM [28]. Other work either generalizes the dense workspace to more expressions [6, 24] and/or to hardware architectures [17, 53]. A sparse workspace was first proposed in [12] to scale SpGEMM to thousands of processors. Subsequent work also generalizes sparse workspaces to more expressions [2, 57] and various architectures [8, 58, 82].

Current sparse library frameworks also leverage workspaces. Eigen [26] uses optimized Gustavson's dense workspace by converting the input matrices to be compatible with row-wise SpGEMM. CTF [67] reduces tensor contractions to matrix multiplications and transpositions where each sub-tensor result works as a workspace that is reduced in the end.

Unlike these prior workspace algorithms and library frameworks, our compiler approach is general for any tensor algebra expression and modular across various workspace designs. The design of our compiler allows for optimization techniques and core data structure strategies while abstracting away architecture-specificity.

Workspace Optimizations in Sparse Compilers. This paper integrates the sparse workspace algorithm into an existing sparse compilation framework, enabling it to generate code for arbitrary tensor algebra expressions that have sparse scattering. Early work on sparse compilers only had first-order and second-order dense workspaces for linear algebra. Bik and Wijshoff [10] materialize the internal sparse collection as a dense workspace in their restructuring compiler for row-wise and outer-product SpGEMM. In SIPR [62], a dense workspace is implemented for row-wise SpGEMM as a C++ class with access and update methods. Recent work on sparse compilers generalize dense workspaces to tensor algebra [9, 38, 44] using schedules [13, 38, 63, 65]. Workspaces are indispensable for the scattering problem, therefore, sparse compilers without workspaces can only support appending expressions [83, 86, 87].

Sparse Tensor Format Abstraction. The sparse workspace techniques in this work bridge the gap between prior techniques on sparse iteration and tensor format conversions to solve the sparse scattering problem. Without format conversions, the sparse workspace would be unable to generate the result tensor format assigned by the user. Researchers have been studying sparse tensor formats and efficient format conversion algorithms for decades in order to utilize the data distribution for optimizing specific expressions [12, 28, 35, 64, 77, 78]. Early sparse compilers have their own individual format systems [5, 43, 75], but recent work on compilers generalize these approaches by creating a uniform representation and conversion routine for common compressed formats [14, 15, 51]. The sparse workspace tensor format described in our work (SpFormat) for ISM arrays is defined using the same level format abstraction presented by Chou et al. [14], and the code generation for the compression stage of our compiler leverages work from Chou et al. [15].

9 CONCLUSION

We introduce a compiler for sparse tensor algebra that can generate code for expressions with sparse scattering behavior through the introduction of sparse workspaces. Users may either manually insert sparse workspaces using our compiler or rely on the compiler to automatically detect sparse scattering and insert these workspaces. The compiler leverages a four-stage algorithm template, called insert-sort-merge, to abstractly represent a sparse workspace, and we provide data structures and optimizations that implement concrete sparse workspace policies using this algorithm template. Our work extends the generality of prior compilers for sparse tensor algebra by treating sparse tensors as a first-class concept for any tensor variable, including temporaries.

ACKNOWLEDGMENTS

We thank Alexander J. Root, Benjamin Driscoll, Bobby Yan, Chris Gyurgyik, David Broman, Guohao Dai, Kai Zhong, James Dong, Praneeth Kolichala, Rubens Lacouture, Scott Kovach, Shiv Sundram, Shulin Zeng, Rohan Yadav, Yu Wang and Zhenhua Zhu for their helpful feedback. This work was supported in part by the Semiconductor Research Corporation (SRC) PRISM center. Olivia Hsu was supported by an NSF GRFP Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

REFERENCES

[1] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. 269–285.

[2] Kadir Akbudak and Cevdet Aykanat. 2014. Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 36, 5 (2014), C568–C590.

[3] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced hashing and efficient GPU sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. (2024).

[5] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and verifying sparse matrix codes. *ACM Sigplan Notices* 45, 9 (2010), 249–260.

[6] Ariful Azad and Aydin Buluç. 2017. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 688–697.

[7] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 394–419.

[8] Nathan Bell, Steven Dalton, and Luke N. Olson. 2012. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[9] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (sep 2022), 25 pages. https://doi.org/10.1145/3544559

[10] Aart JC Bik and Harry AG Wijshoff. 1994. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing: 6th International Workshop Portland, Oregon, USA, August 12–14, 1993 Proceedings 6*. Springer, 57–75.

[11] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures* (Hilton Head, South Carolina, USA) *(SPAA '91)*. 3–16.

[12] Aydin Buluc and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[15] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. 823–838.

[16] Donald B Cleveland, Ana D Cleveland, and Olga B Wise. 1984. Less than full-text indexing using a non-boolean searching model. *Journal of the American Society for Information Science* 35, 1 (1984), 19–28.

[17] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the GPU. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

[18] Julien Demouth. 2012. Sparse matrix-matrix multiplication on the GPU. In *Proceedings of the GPU technology conference*, Vol. 3.

[19] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium*

Workshops (IPDPSW). IEEE, 693–702.

[20] Luc Devroye. 1986. Lecture notes on bucket algorithms. *Progress in computer science* 6 (1986).

[21] Dawson R. Engler. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. 160–170.

[22] M.A. Ertl and D. Gregg. 2004. Retargeting JIT compilers by using C-compiler generated executable code. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 41–50.

[23] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. IEEE, 1381–1384.

[24] 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. https://doi.org/10.1137/0613024 arXiv:https://doi.org/10.1137/0613024

[25] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.

[26] Gaël Guennebaud, Benoit Jacob, et al. 2010. Eigen. *URL: http://eigen.tuxfamily.org* 3 (2010).

[27] Luanzheng Guo and Gokcen Kestor. 2023. On Higher-performance Sparse Tensor Transposition. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 697–701.

[28] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (sep 1978), 250–269. https://doi.org/10.1145/355791.355796

[29] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. arXiv:2212.11142 [cs.PL]

[30] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (oct 2021), 29 pages. https://doi.org/10.1145/3485505

[31] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726.

[32] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. 703–718.

[33] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*. 1047–1058. https://doi.org/10.1109/ICDE.2015.7113355

[34] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719918

[35] David R Kincaid, Thomas C Oppe, and David M Young. 1989. *ITPACKV 2D user's guide.* Technical Report. Texas Univ., Austin, TX (USA). Center for Numerical Analysis.

[36] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[37] Fredrik Kjolstad. 2020. *Sparse Tensor Algebra Compilation.* Ph.D. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://tensor-compiler.org/files/kjolstad-phd-thesis-taco-compiler.pdf

[38] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. (2019), 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

[39] 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

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

[41] Tamara G. Kolda and Jimeng Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *2008 Eighth IEEE International Conference on Data Mining*. 363–372. https://doi.org/10.1109/ICDM.2008.89

[42] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. 2019. The suitesparse matrix collection website interface. *Journal of Open Source Software* 4, 35 (2019), 1244.

[43] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing: Third International Euro-Par Conference Passau, Germany, August 26–29, 1997 Proceedings 3*. Springer, 318–327.

[44] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (jun 2023), 25 pages.
[45] Daniel Langr, Pavel Tvrdik, and Ivan Simecek. 2016. AQsort: Scalable multi-array in-place sorting with OpenMP. *Scalable Computing: Practice and Experience* 17, 4 (2016), 369–391.
[46] Valentin Le Fèvre and Marc Casas. 2023. Efficient Execution of SpGEMM on Long Vector Architectures. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3588195.3593000
[47] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. IEEE, 26–33.
[48] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 806–814.
[49] Joseph W. H. Liu. 1992. The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Rev.* 34, 1 (1992), 82–109. https://doi.org/10.1137/1034004 arXiv:https://doi.org/10.1137/1034004
[50] Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 370–381.
[51] Suzanne Mueller, Peter Ahrens, Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Sparse Tensor Transpositions. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. 559–561.
[52] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2020. Comet: A domain-specific compilation of high-performance computational chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 87–103.
[53] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-Performance Sparse Matrix-Matrix Products on Intel KNL and Multicore Architectures. In *Workshop Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP Workshops '18)*. Article 34, 10 pages.
[54] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.
[55] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.
[56] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106.
[57] Carlos Ordonez, Yiqun Zhang, and Wellington Cabrera. 2016. The Gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering* 28, 7 (2016), 1905–1918.
[58] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.
[59] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. 362–375.
[60] 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.
[61] Christoph Pichler, Paley Li, Roland Schatz, and Hanspeter Mössenböck. 2023. Hybrid Execution: Combining Ahead-of-Time and Just-in-Time Compilation. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Cascais, Portugal) *(VMIL 2023)*. 39–49.
[62] William Pugh and Tatiana Shpeisman. 1998. SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC '98)*. Springer-Verlag, Berlin, Heidelberg, 213–229.
[63] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530.
[64] Yousef Saad. 2003. *Iterative methods for sparse linear systems.* SIAM.

[65] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

[66] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional Collection Programming with Semi-Ring Dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (apr 2022), 33 pages.

[67] Navjot Singh, Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. 2022. Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels. *J. Parallel and Distrib. Comput.* 169 (Nov. 2022), 269–285. https://doi.org/10.1016/j.jpdc.2022.07.005

[68] James E Smith. 1982. Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News* 10, 3 (1982), 112–119.

[69] Shaden Smith, Alec Beri, and George Karypis. 2017. Constrained Tensor Factorization with Accelerated AO-ADMM. In *2017 46th International Conference on Parallel Processing (ICPP)*. 111–120. https://doi.org/10.1109/ICPP.2017.20

[70] 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.* http://frostt.io/

[71] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 1–7.

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

[73] James M. Stichnoth and Thomas Gross. 1997. Code Composition as an Implementation Language for Compilers. In *Conference on Domain-Specific Languages (DSL 97)*. USENIX Association, Santa Barbara, CA.

[74] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[75] Scott Thibault, Lenore Mullin, and Matt Insall. 1994. *Generating indexing functions of regularly sparse arrays for array compilers.* Technical Report. Technical Report, University of Missouri, Rolla, 1994, TR 94-08.

[76] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the evolution of user interaction in Facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks (WOSN '09)*. 37–42.

[77] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel transposition of sparse data structures. In *Proceedings of the 2016 international conference on supercomputing*. 1–13.

[78] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the GPU. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 68–81.

[79] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. 2023. Unified Convolution Framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems* 5 (2023).

[80] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[81] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 286–300.

[82] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2023. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 598–610.

[83] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. New York, NY, USA, 660–678.

[84] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. arXiv:2404.04541 [cs.PL]

[85] Genghan Zhang, Yuetong Zhao, Yanting Tao, Zhongming Yu, Guohao Dai, Sitao Huang, Yuan Wen, Pavlos Petoumenos, and Yu Wang. 2023. Sgap: towards efficient sparse tensor algebra compilation for GPU. *CCF Transactions on High Performance Computing* (2023), 1–18.

[86] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. 2022. Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration. *ACM Trans. Archit. Code Optim.* 20, 1, Article 16 (Dec 2022), 26 pages. https://doi.org/10.1145/3566054

[87] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. 2022. SparTA: Deep-Learning Model Sparsity via Tensor-with-Sparsity-Attribute. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 213–232.

Received 2023-11-16; accepted 2024-03-31