Compiler Support for Sparse Tensor Convolutions

PEIMING LIU, Google Research, USA
ALEXANDER J ROOT, Stanford University, USA
ANLUN XU, Google Cloud, USA
YINYING LI, Google Research, USA
FREDRIK KJOLSTAD, Stanford University, USA
AART J. C. BIK, Google Research, USA

This paper extends prior work on sparse tensor algebra compilers to generate asymptotically efficient code for tensor expressions with affine subscript expressions. Our technique enables compiler support for a wide range of sparse computations, including sparse convolutions and pooling that are widely used in ML and graphics applications. We propose an approach that gradually rewrites compound subscript expressions to simple subscript expressions with loops that exploit the sparsity pattern of the input sparse tensors. As a result, the time complexity of the generated kernels is bounded by the number of stored elements and not by the shape of the tensors. Our approach seamlessly integrates into existing frameworks and is compatible with recent advances in compilers for sparse computations, including the flexibility to efficiently handle arbitrary combinations of different sparse tensor formats. The implementation of our algorithm is open source and upstreamed to the MLIR sparse compiler. Experimental results show that our method achieves 19.5x speedup when compared with the state-of-the-art compiler-based method at 99.9% sparsity. The generated sparse kernels start to outperform dense convolution implementations at about 80% sparsity.

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

Additional Key Words and Phrases: convolution, sparse tensors, sparse tensor algebra, sparse data structures, performance, code generation, merge lattices, iteration graphs

ACM Reference Format:
Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart J. C. Bik. 2024. Compiler Support for Sparse Tensor Convolutions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 281 (October 2024), 29 pages. https://doi.org/10.1145/3689721

1 Introduction

Tensors are fundamental abstractions to represent high-dimensional data in many fields of science, engineering, data analytics, and machine learning. Often, these tensors are sparse, i.e., contain many zeros, which offers interesting opportunities for reducing their memory footprint as well as reducing the computation time of operations on such tensors. Exploiting sparsity by hand has been well-studied in the past for sparse matrices arising in linear-algebra problems [14, 21, 44, 51, 59]. Utilizing sparsity in high-dimensional tensors is becoming increasingly important as sparse neural networks are gaining more attention [27]. However, any performance benefit from sparsity comes with an

Authors' Contact Information: Peiming Liu, Google Research, Seattle, WA, USA, peiming@google.com; Alexander J Root, Computer Science, Stanford University, Stanford, CA, USA, ajroot@stanford.edu; Anlun Xu, Google Cloud, Mountain View, CA, USA, anlunx@google.com; Yinying Li, Google Research, New York, USA, yinyingli@google.com; Fredrik Kjolstad, Computer Science, Stanford University, Stanford, CA, USA, kjolstad@stanford.edu; Aart J. C. Bik, Google Research, Mountain View, CA, USA, ajcbik@google.com.

ACM 2475-1421/2024/10-ART281
https://doi.org/10.1145/3689721

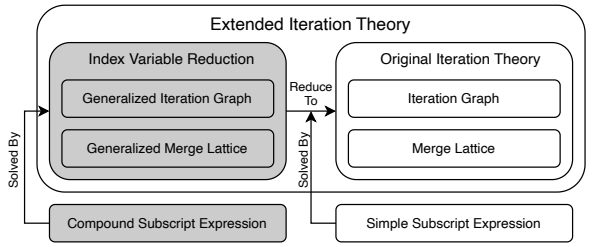


Fig. 1. System overview with contributions of the paper highlighted.

increased complexity in both data representations and algorithmic implementation. Given the wide range of different sparse-tensor formats and possible numeric formulations, it is notoriously difficult to implement a sparse computation properly. Thanks to pioneering research in sparse compilers [4–7] and recent advances to generalize sparse compiler techniques [25, 29–31], a wide range of sparse computations can now be automatically generated by compilers with greater ease, flexibility, and often better performance compared to hand-written sparse code.

Despite these advances, various computations are still not supported by existing sparse-tensor compilers. Among those cases, many of the compilation shortcomings are caused by the use of compound subscript expressions, i.e., subscript expressions using more than one index variable. For example, a 1-D convolution $A_i = \sum_{k=0}^{n} I_{i+k} F_k$ imposes the compound subscript expression $i + k$ on tensor I. Generating efficient code for such computations is difficult because multiple values of index variables need to be determined so as to fetch the desired stored element in a sparse tensor. Generally, each index variable corresponds to a *single* loop and multiple loops are required to determine the value of a compound subscript expression. For dense computations, these loops incur no performance penalty. For sparse computations, on the other hand, the lack of random-access support in most sparse data structures poses a performance challenge. For this reason, existing methods are restricted to trivial index expressions and fail to generate code for the more complex cases.

Other research has explored how to enhance the capability of the existing framework, resulting in compiler support for important kernels like sparse convolution, as presented in TACO-UCF [56]. Our work generalizes over TACO-UCF. Similar to TACO-UCF, our technique generates code for multiple convolution variants. In contrast to TACO-UCF, our work achieves better asymptotic complexity in many cases by exploiting additional optimization opportunities for sparsely iterating over sparse tensors. The differences between the code generated by TACO-UCF and our work are explained in more detail in Section 2.

Our code generation algorithm generalizes the sparse-iteration model [28, 31] beyond simple index expressions. We focus on traditional convolutions as examples and experimental validation in this paper, but our algorithm generalizes to a much wider class of computations and offers the same level of flexibility as TACO to allow arbitrary combinations of different operations on sparse tensors, allowing us to handle much more complicated tensor index expressions such as:

- B_{i+j+k}, i.e., more than two index variables in a single subscript expression,
- $A_{i+j} B_{i+k}$, i.e., more than one input tensor with compound subscript expressions,
- A_{2i+j}, i.e., subscript expressions with constant coefficients on index variables,
- and arbitrary combinations of the above cases.

The technical contributions of this paper to the field of sparse compilation are the following:

Naive	TACO-UCF	MLIR (This work)

`// dense outer loop`
`for (int i = 0; i < n; i++) {`
`  // dense inner loop`
`  for (int j = 0; j < k; j++)`
`  locate(C, i + j);`
`  ...`
`}`
$O(n \cdot k \cdot log(nnz(C)))$
	`// dense outer loop`
	`for (int i = 0; i < n; i ++) {`
	`  auto [lo, hi] = searchPos(..);`
	`  // sparse inner loop`
	`  for(; lo < hi; lo++)`
	`  ...`
	`}`
	$O(n \cdot \frac{nnz(C) \cdot k}{n+k})$
		`// sparse outer loop`
		`for (auto w : nonEmptySubsec(C,k)) {`
		`  int i = w.offset, lo = w.lo;`
		`  // sparse inner loop`
		`  for(; lo < w.hi; lo++)`
		`  ...`
		`}`
		$O(min(n, k \cdot nnz(C)) \cdot \frac{k \cdot nnz(C)}{n+k})$

Fig. 2. Three implementations of the 1D sparse convolution $A_i = \sum_{j=0}^{n} C_{i+j} B_j$ with different asymptotic complexity. C is sparse, B is dense, $i \in [0, n)$, and $j \in [0, k)$.

- A generalized theory of sparse iteration that supports compound affine subscript expressions on sparse tensors.
- A new code generation algorithm that retains the sparsity pattern of the input tensor, making compiler generated sparse convolution kernel practical for a wide range of sparse storage formats.

Figure 1 shows an overview of our approach. The improvements over existing methods are highlighted. Our generalization unifies the theory for handling trivial and non-trivial subscript expressions, making the original sparse tensor algebra compilation work [31] the base case under the new framework. Conceptually, our method gradually reduces non-trivial subscript expressions to trivial ones by *index variable reduction*. Our implementation is made publicly available as a part of the sparse compiler technology [3] embedded in the MLIR compiler infrastructure [36]. We compared our system with the state-of-the-art compiler-based approach, TACO-UCF [56], and show that the proposed algorithm, while performs equally well on the set of sparse formats that TACO-UCF excels, is able to handle more sparse tensor formats efficiently, especially formats used to compress very sparse tensors. Our experimental results show a $2\times - 19\times$ runtime speedup at $90\% - 99.9\%$ sparsity.

2 Motivation and Background

This section presents a sparse 1-D convolution kernel ($A_i = \sum_{j=0}^{n} C_{i+j} B_j$) as a running example to provide a high level explanation of our approach. Later sections will discuss how the framework can be extended to handle high-dimensional tensors and more complex expressions.

Since dense arrays support random accesses, i.e., values indexed by arbitrary coordinates can be located and loaded in constant time, the implementation to extract values from dense arrays with compound subscript expression is straightforward. However, most sparse tensor formats do not support such constant-time lookup. As the result, problems arise when multiple index variables are used in a single compound subscript expression, where multiple loops are required to resolved the expression.

As shown in Figure 2, a naive code generation algorithm to iterate over the values of C_{i+j} would simply generate two dense loops that independently iterate over i and j. The resulting sparse kernel has a time complexity that is *worse* than the corresponding dense computation, due to an extra $O(log(nnz))$ binary search for locating each value. TACO-UCF improves on the naive approach by only generating a dense loop for the outer index variable, using a sparse inner loop to iterate over subsections of the sparse input, as shown in the center of Figure 2. By utilizing the fact that i is monotonically increasing, the starting position can be found with `searchPos` in amortized $O(1)$ time [56]. While this implementation is far more efficient than the naive approach with two dense loops, the resulting kernel is still asymptotically sub-optimal because of the dense outer loop

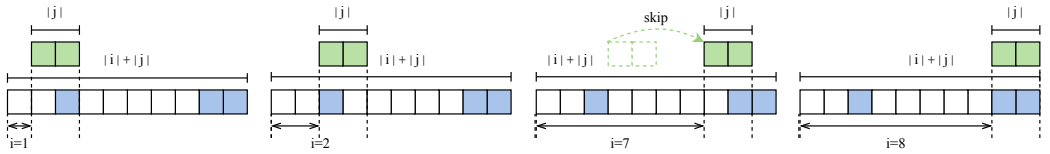


Fig. 3. Illustration of visited elements by the code generated by our algorithm.

around the locate operation. Our approach, as shown on the right side of Figure 2, goes further by transforming the problems into finding and iterating over a *sparse* set of *non-empty* subsections*, with i being reduced to the *offset* of the subsection being visited. This difference leads to a reduction from $O(n)$ to $O(min(n, k \cdot nnz(C)))$†. For practical convolution kernels, the typical value of k is tiny (e.g., $k = 3$), so $k \cdot nnz(C) \ll n$ when C is very sparse.

Now consider running the same 1-D convolution $A_i = \sum_{j=0}^{n} C_{i+j} B_j$ over a sparse vector as shown in Figure 3 with $|j| = 2$, $|i| = 8$ (i.e., $i \in [0, 2)$, $j \in [0, 8)$). To locate a value at $i + j$ in C_{i+j}, both values of i and j must first be determined. Suppose i is iterated over first, since it is used in a compound subscript expression, the *sparse* set of values to traverse for i are no longer equal to the set of stored coordinates in the sparse tensor. In Figure 3, since the set of stored coordinates is $\{2, 8, 9\}$ and the convolution kernel has a width of two, i should iterate over the coordinate values $\{1, 2, 7, 8\}$. As we elaborate more in Section 3.1, conceptually, the set of value to visit for a reduced index variables from compound subscript expressions can be viewed as the set of *offsets* of non-empty subsections that can be extracted from the sparse tensor. So iterating over i in $i + j$ is equivalent of iterating over the set of *non-empty* subsections with size equal to $|j| = 2$. The value of i is then determined by the offset of the current subsection. Figure 3 shows that the set of non-empty subsections of size 2 is $\{C[1:3], C[2:4], C[7:9], C[8:10]\}$, and thus the corresponding offset set $\{1, 2, 7, 8\}$ is the set of valid values for i with the given input. Unlike TACO-UCF, which iterates over every $i \in [0, 8)$, our approach skips $i \in [3, 7)$ and thus has a better performance. For very sparse data, the set of skipped values of i becomes greater than the set of visited values. The paper presents the first systematic compiler-based approach to specify (Section 3.2) and traverse (Section 5.2) *only non-empty* subsections in a sparse tensor of various formats. Note that non-empty subsections can also be extracted outside the compute loop nest and/or added to a list that can then be iterated densely. Such extraction allows a further loop unrolling optimization (if desired) that turns the 1-D convolution in Figure 3 into a sequence of sparse element-wise multiplication over each non-empty subsection (e.g., $O_1 = \sum_{j=0}^{k} C[1:3]_j \times F_j$ for the first subsection).

After i is reduced from the subscript expression $i + j$, the problem becomes simple enough to be handled by existing approaches. For instance, when i is reduced to $i = 1$, the problem is reduced to $A_1 = \sum_{j=0}^{n} C[1:3]_j \times B_j$ with only a simple subscript expression. Standard convolutions using multiplication (a conjunctive operator) require both operands to be non-zero to produce non-zero results. Disjunctive operators such as addition result in multiple cases where the result is non-zero if *any* operand is non-zero. For example, in $A_i = \sum_{j=0}^{n} C_{i+j} + B_j$ with sparse C and dense B, the loops need to handle multiple cases depending on whether $C[1:3]$ is empty after i is reduced to $i = 1$. That is, both cases of $A_1 = \sum_{j=0}^{n} C[1:3]_j + B_j$ and $A_1 = \sum_{j=0}^{n} B_j$ need to be handled. Both

*A.k.a., tensor slices. We use the term "tensor subsections" in this paper to avoid confusion since we do not actually extract tensor slices but use it as the abstraction to explain the algorithm.

†The expression estimates the maximum number of non-empty subsections of size k on C. Since each non-zero element can *at most* be included by k subsections, hence $O(k \cdot nnz(C))$. In addition, the number of non-empty subsections on C of size k is bounded by n in the worst case, hence $O(min(n, k \cdot nnz(C)))$.

disjunctive and conjunctive operators are handled in our code generation algorithm (see Section 5.1 for more details).

3 An Iteration Model For Compound subscript expressions

Figure 4 defines a language grammar for tensor index notation expressions [31] that allows compound subscript expressions. Our grammar introduces a new definition called an *index term* that allows optional coefficients on index variables, and a new definition called *subscript expression* to allow compound affine expressions for tensor subscripts. In this section, we show how a compound subscript expression can be *reduced* to a trivial one using *non-empty subsection driven iterations*.

3.1 Non-empty Subsection-driven Iteration

Section 2 informally explained how loops for a reduction of index variable i from a compound subscript expression $i + j$ can be transformed into finding a set of non-empty subsections of size $|j|$. In this section, we extend the idea to handle multi-dimensional tensors and introduce the new loop primitives to construct loops over non-empty subsections.

3.1.1 Subsection driven loops, a 2-D example: Figure 5 illustrates an abstraction of the code generated for iterating over the compound subscript expression $T_{i+k,j+l}$. As shown in the figure, to construct non-empty subsection driven loops, we introduce the following loop primitives (the code generation algorithm for each individual primitive is described in Section 5):

- `S = subsect_begin(T, D, Z)` to extract the first non-empty subsection `S` (with the smallest offset) from `T` along dimension `D` with size `Z`.
- `S = subsect_next(S)` to forward the current subsection `S` and find the next non-empty subsection with a larger offset.
- `subsect_offset` to extract the offset of the subsection.
- `subsect_end(S)` to terminate loops if the current subsection is empty (and there is no remaining subsection to visit).

Conceptually, since each index variable represents a range of data to be visited by a loop, to handle index variables i, k, j, l used in $T_{i+k,j+l}$, we need to generate four nested loops as shown in Figure 5. Depending on the user-selected loop order, non-empty subsection driven loops can be placed at different depths, resulting in different access patterns. In the remaining of the paper, we say an index variable i is **reduced** from a subscript expression (or the corresponding loop of i is an *index reduction loop*) if generating a loop for i does not determine the value of the entire subscript expression (i.e., i is not the *last* index variable to be handled in the subscript expression per user-selected loop generating order). Otherwise, we say i **resolves** the subscript expression (or the corresponding loop of j is an *index resolve loop*). For example, when using loop order $i \rightarrow j$ (i.e.,

$$
\begin{array}{rcl@{\qquad}rcl}
\text{assigment} & := & tensor_{indices} = expr & \text{operation} & := & -expr \mid expr + expr \\
\text{expr} & := & reduction \mid operation & & \mid & expr - expr \mid expr \times expr \\
& \mid & literal \mid access & \text{indices} & := & index \mid index, indices \\
& \mid & (expr) & \text{reduction} & := & \sum_{indices} expr \\
\text{subscript} & := & idx\_term + subscript & \text{access} & := & tensor_{subscript^*} \\
& \mid & idx\_var & \text{idx\_term} & := & literal \mid literal \times index \mid index \\
\text{index} & := & i \mid j \mid \ldots & \text{literal} & := & 1 \mid 2 \mid \ldots
\end{array}
$$

Fig. 4. The extended grammar of tensor index notation [31]. Items in blue are our extensions to handle non-trivial subscript expression.

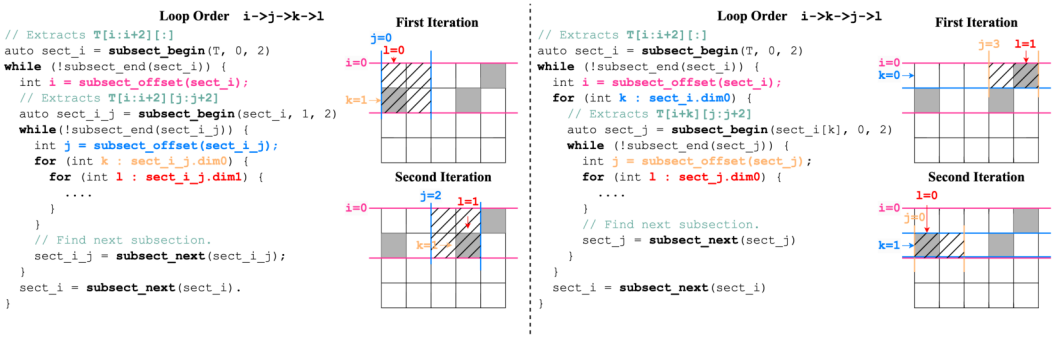


Fig. 5. The non-empty subsection driven loops used to traverse a 2-D matrix with compound subscript expression as in $T_{i+k,j+l}$ (supposing $k \in [0, 2)$ and $l \in [0, 2)$). Figures on the left and right show different loop orders (schedules) that lead to different strategies to extract non-empty subsections. ■ marks the non-zeros in the matrix. Shadowed area marks the smallest subsections extracted by the innermost subsection driven loop.

i is an outer loop of j) to handle the subscript expression $i + j$, loop i is an index reduction loop and loop j is an index resolve loop. Only index reduction loops iterate over non-empty subsections and index resolve loops directly iterate over stored non-zeros.

Figure 5 shows two potential loop orders to traverse $T_{i+k,j+l}$. For the loop order ($i \rightarrow j \rightarrow k \rightarrow l$) used in Figure 5 (left), the two outermost loops that handle index variable i and j are subsection driven loops because of the remaining unhandled index variable k in $i + k$ and l in $j + l$ respectively. The outermost loop for i iterates over every non-empty subsection `T[i,i+2][:]` and the second loop refines the current subsection to include only `T[i,i+2][j:j+2]`. At the first iteration, the two subsection driven loops locate a 2×2 subsection at `T[0:2][0:2]` (i.e., $i = 0, j = 0$). The following two inner loops then iterate over the extracted subsection `T[0:2][0:2]` and locate the first (and only) non-zero included in the subsection at `T[0+1][0+0]` (i.e., $k = 1, l = 0$). After exhausting all the non-zeros in the subsection, `T[0:2][0:2]` is then forwarded to `T[0:2][2:4]` (i.e., $i = 0, j = 2$) in the next iteration.

On the other hand, for the loop order ($i \rightarrow k \rightarrow j \rightarrow l$) used in Figure 5 (right), the two outermost loops handle index variable i and k from the same subscript expression $i + k$. As the result, the subsection required to reduce j from $j + l$ in the third loop is extracted from a single row of `T` (i.e., `T[0+0]` at the first iteration), which leads to a 1-D subsection `T[0+0][3:5]` to be extracted (i.e., $j = 3$). After exhausting all non-zeros in `T[0+0][3:5]` by the innermost loop, k is then forwarded the second non-empty row `T[0+1]` and $j = 0$ is used to extract the next subsection `T[0+1][0:0+2]` to be iterated over.

3.1.2 Proof of correctness. In this paragraph, we formalize the problem transformation (i.e., transforming a reduction of index variable from compound subscript expressions into subsection driven loops). We also generalize the problem to allow arbitrarily complex subscript expressions $i+j+k+\ldots$ ($i \in [0, |i|), j \in [0, |j|), k \in [0, |k|), \ldots$ [‡] and prove the correctness of the transformation.

To efficiently reduce i from $C_{i+j+k+\ldots}$, an *optimal* algorithm should only iterate over the set of *constrained values* of i' if and only if there exists an index idx, and values $j, k + \ldots$, such that $idx = i' + j + k + \ldots$, and idx locates a non-zero element in C. Formally, with the remaining *free* index

[‡]In our implementation, the bound of each index variable is inferred by MLIR `linalg` dialect in a similar way as described by tensor comprehensions [53].

variables set $\mathcal{F} = \{j \in [0, |j|), k \in [0, |k|), ...\}$, the set of constrained value $\mathcal{I}$ that should be iterated to *optimally* reduce i can be described as $\mathcal{I} = \{i' \mid \exists j \in [0, |j|), \exists k \in [0, |k|), ..., C[i'+j+k+...] \neq 0\}$.

On the other hand, we say a subsection $C[o : o + s]$ extracted from C is *non-empty* if and only if $C[o : o + s]$ covers at least one specified element stored in C (i.e., $\exists idx \in [0, s), C[o + idx] \neq 0$). By extracting subsections with size equals to the sum of bounds of remaining free index variables (i.e., $s = \sum_{f \in \mathcal{F}} |f|$) from C and denoting the set of non-empty subsections with size s as $\mathcal{N}_c = \{C[o : o + s] \mid o \in [0, |i|) \wedge \neg empty(C[o : o + s])\}$, we can formally prove the correctness of the problem transformation by proving the following theorem.

Theorem 3.1. $C[i' : i' + s] \in \mathcal{N}_c \Leftrightarrow i' \in \mathcal{I}$

Proof. $C[i' : i'+s] \in \mathcal{N}_c \Rightarrow \neg empty(C[i' : i'+s])$. By the definition of the non-empty subsection, we have ❶ $\exists s' \in [0, |s|) \Rightarrow C[i' + s'] \neq 0$. On the other hand, since $s = \sum_{f \in \mathcal{F}} |f| = |j| + |k| + \ldots$, we know that ❷ $\exists j \in [0, |j|), \exists k \in [0, |k|), \ldots$ such that $s' = j + k + ...$ Combining ❶ and ❷, we know that $\exists j \in [0, |j|), \exists k \in [0, |k|), ..., \Rightarrow C[i' + j + k + ...] \neq 0$, thus, by the definition of $\mathcal{I}$, $i' \in \mathcal{I}$. The reverse can be proven similarly. □

Theorem 3.1.2 forms the foundation for reducing index terms from compound subscript expressions by generating non-empty subsection driven loops. To extend Theorem 3.1.2 to handle an index term with a constant coefficient c, the non-empty subsection can be specified by an additional $stride = c$.

3.2 Sparse Subsection Specification

In this section, we describe a detailed approach that can precisely specify a multi-dimensional subsection from a sparse tensor. As we show later in Section 5, we can efficiently construct the described data structure required to specify a subsection *on-the-fly* during iteration.

The storage of a sparse tensor, as the 2-D matrix shown in Figure 6 (a), can conceptually be viewed as a tree, where each tree level represents a dimension of the sparse tensor. Meanwhile, a sparse tensor subsection can be viewed as a sub-tree of the original storage tree as in Figure 6 (b). Different sparse tensor formats encode the tree structure in different ways. Using the same data representation as in [29] as an example, for a `compressed` dimension, it requires two integer arrays, namely `position` and `index` array. They together form a segmented vector, with one segment (a list of child nodes) per entry in the previous dimension (the parent node). The `index` array stores the indices (coordinates) of specified elements in that dimension, while the `pos` array stores the position range for the corresponding segment in the next dimension. That is, segment `i` is located by `pos[i]` and `pos[i+1]`. For example, as in Figure 6, the column segment for $\texttt{index}_{row}\texttt{[1]}$ is specified by the range [$\texttt{pos}_{col}\texttt{[1]}$, $\texttt{pos}_{col}\texttt{[2]}$).

To specify a sparse tensor subsection, a similar structure can be constructed to specify a subtree from the original storage tree. In our technique, it is achieved by:

- memorizing the subsection metadata, i.e., a pair of `(offset, size)`, for each dimension, and
- maintaining a separate `position` array that only includes the coordinates in the subsection.

As shown in Figure 6 (c) and (d), we store 3 instead of 2 for the lower bound of the second segment for column because $(\texttt{index}_{col}\texttt{[2]} = \texttt{0}) < \texttt{offset}_{col}$ (thus should not be included in the subsection).

In our technique, we allow *loose* upper bounds for the subsection position range, meaning that the range could contain indices that exceed the subsection boundary. As shown in Figure 6 (c), the position range for the first dimension of `A[0:2][1:3]` is $[0, 3)$ despite that $2 \in [0, 3)$ and $\text{index}_{row}[2] = 3$ exceeds the boundary of A[0:2] (since $3 \geq 2$). This is because:

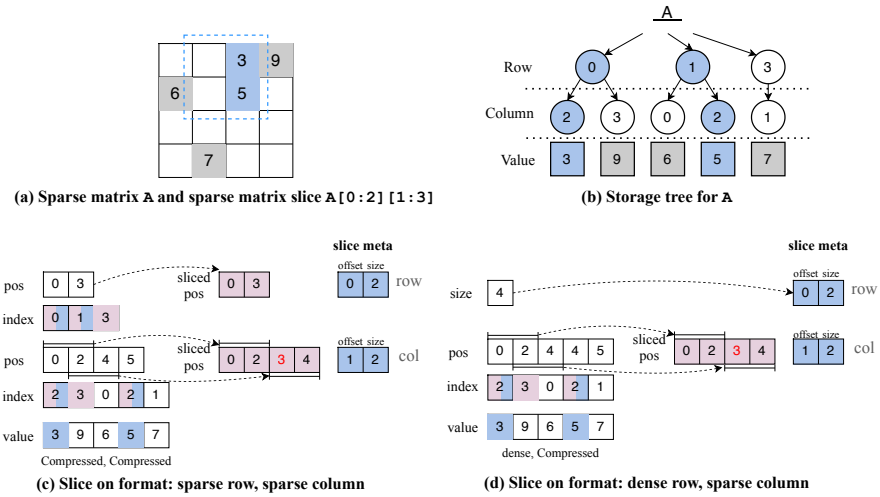


Fig. 6. Example showing how a subsection can be extracted from a sparse tensor storage tree. ■ marks the elements that are included by pairs of position low and high bounds; ■ marks the elements that are actually included in the subsection specified by the metadata (i.e., offset and size).

.

(1) Finding the tight upper bound requires a forward lookup on the index array to locate the first out-of-bound coordinate, which can be expensive.
(2) We can still *precisely* iterate only in-bound coordinates without the tight upper bound by generating a `break` statement upon hitting the first out-of-bound coordinate.

To take a subsection over a dense dimension, a pair of `(offset, size)`, as shown in Figure 6 (d), is sufficient.

4 Generalized Iteration Graph

An iteration graph defines the loop order for iterating over stored elements of sparse tensors used in a single tensor expression. In the original definition [31], an iteration graph is composed of a set of *index variables* and a set of *tensor paths*, where each index variable represents the iteration over one level of a tensor, or co-iteration over levels from multiple tensors (when the same index variable is used by multiple tensors in the tensor expression). To handle non-trivial subscript expressions, we introduce the concept of an *unresolved index term set*.

Definition 4.1. An *unresolved index term set* of a tensor level T_l is a *subset* of the set of index terms used in the subscript expression on T_l. A tensor level T_l, together with its unresolved index term set is denoted in the form of $\mathcal{S}_{T_l} : \{i, j, k\}$.

The *initial* value of $\mathcal{S}_{T_l}$ can be determined by extracting all index terms used in the subscript expression on T_l. For example, given the tensor expression $A_{i,j} = I_i J_{i,j+2}$, we have $\mathcal{S}_{I_0} : \{i\}$, $\mathcal{S}_{J_0} : \{i\}$ and $\mathcal{S}_{J_1} : \{j, 2\}$ initially. At a high level, $\mathcal{S}_{T_l}$ specifies the set of *free* index terms whose values remain to be determined at the current code generation stage, that is, loops to resolve those index terms have not yet been generated. By only admitting tensor expression with $|\mathcal{S}_{T_l}| = 1$, we fall back to the original framework in which only trivial subscript expressions can be handled.

Definition 4.2. A *tensor path* $p^* = \{(v_i, \mathcal{S}_{T_l}), ...\}$ is a list of index variables v_i that are annotated with an unresolved index term set $\mathcal{S}_{T_l}$.

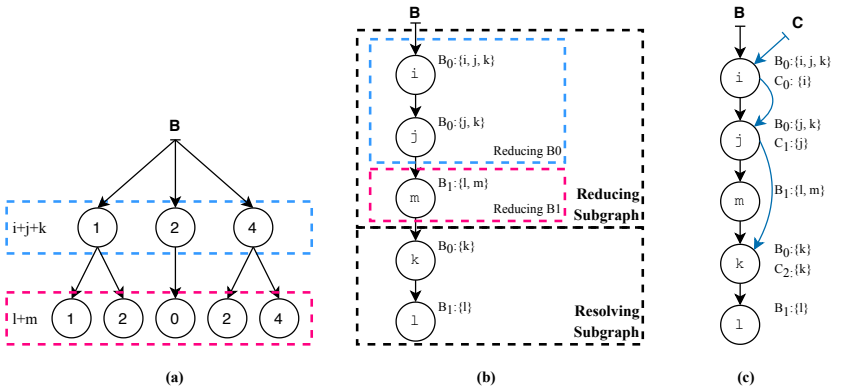


Fig. 7. Figure **(a)** shows the conceptual storage tree for a 2-D tensor. Figure **(b)** shows the iteration graph for tensor expression $B_{i+j+k,l+m}$. A tensor path through an iteration graph summarizes how each index variable in non-trivial index expressions are reduced to trivial ones (during reducing phase) before the tensor storage is traversed from root to leaves (during resolving phase). Figure **(c)** shows a merged iteration graph for tensor expression $B_{i+j+k,l+m}C_{i,j,k}$.

Definition 4.3. An *iteration graph* is a directed graph $G = (V, P^*)$ with a set of index variables $V = \{v_i, v_j, ...\}$ and a set of extended tensor path $P^* = \{p_i^*, p_j^*, ...\}$.

Figure 7 shows how an iteration graph can be constructed from a tensor expression with non-trivial index expressions. As in figure 7 (b), the path to access tensor B is annotated with an additional $\mathcal{S}_{B_i}$ at each node. Starting with the *initial* set collected from the tensor expression, the elements in $\mathcal{S}_{B_i}$ are gradually reduced after passing through vertices in the tensor path. For example, $\mathcal{S}_{B_0} = \{i, j, k\}$ is reduced to $\{j, k\}$ after visiting vertex i in the graph. The reduction of an index variable signifies that the value of the index variable is determined. As such, the index variable becomes invariant in any deeper nested loop. Analogous to the original iteration graph definition [31], each node in the extended iteration graph corresponds to a loop in a loop nest to coiterate over levels from different tensors.

We can conceptually partition the iteration graph into two subgraphs, namely the *reducing* and *resolving* subgraph, by examining the number of remaining index terms in the current unresolved index term set. We say that the subscript expression imposed on tensor dimension T_l is *fully reduced* at a vertex of the iteration graph *iff* $|\mathcal{S}_{T_l}| = 1$, meaning that the compound subscript expression on the dimension has been reduced to a trivial one. In figure 7 (b), the reducing phase refers to all stages *before* vertex k because they all have more than one remaining index variables in corresponding $\mathcal{S}_{B_i}$, and the remaining stages together are considered resolving phase. Similarly, we say that the subscript expression on a tensor dimension T_l is *fully resolved* at a vertex of the iteration graph *iff* $\mathcal{S}_{T_l} = \emptyset$, meaning that the result of its subscript expression is resolved to a known value (it locates to a specific node in the tensor tree storage at that level). In figure 7 (b), B_0 is fully resolved *after* vertex k, and B_1 is fully resolved after vertex l. Moreover, we call the incoming edge on a fully reduced vertex a *resolving edge*, e.g., $m \rightarrow k$ and $k \rightarrow l$ in figure7 (b), and we call the rest *reducing edges*. Under the new definition, it is clear that the original framework can only handle iteration graph with only resolving edges (the resolving subgraph equals to the iteration graph itself).

Extended iteration graphs (with or without compound subscript expressions) can also be seamlessly merged to construct new iteration graphs for more complex tensor expressions (i.e., to handle multiple tensor operands). It is important to note that our framework does *not* restrict the types

of edges being merged. That is, we allow merging arbitrary edges together regardless of whether they are reducing or resolving edges. Merging reducing and resolving edges signifies that the coiteration over sparse tensor dimensions (viz. resolving edges) need to satisfy additional *index reduction constraints* (viz. reducing edges) as described in Section 3.1.2. In the example shown in figure 7 (c), merging B and C over i means to iterate C_0 with an additional constraint imposed by the reducing edge from B_0, i.e., to iterate C only when $\exists j \exists k (i' + j + k) \in \{1, 2, 4\}$, in which i' is the concrete value of i at the current iteration, and $\{1, 2, 4\}$ are the specified elements in the first level of B as shown in figure 7 (a).

The iteration graph can be constructed automatically for a given tensor expression. Tensor paths with only resolving edges are constructed in the same way as in TACO [31]. To handle tensor paths with reducing edges, we pick an arbitrary feasible index variable p from the unresolved index variable sets from each tensor level. The selected index variable represents the *last* index variable to be resolved, and is used to ensure that levels of the same tensor are resolved in the same order as its storage tree. In principle, the index variable resolving order can be picked arbitrarily. That is, for index expression $i + j$, both $i \rightarrow j$ and $j \rightarrow i$ can be admitted and our code generation algorithm can also handle "interleaved" paths such as $B_0\{i, k\} \rightarrow B_1\{j, l\} \rightarrow B_0\{k\} \rightarrow B_1\{l\}$, in which the two edges for B_0 are interleaved by another edge for B_1. In practice, we found that different orders significantly affect the performance of the generated code (as shown and analyzed in our experimental results). In this paper, we focus mostly on the code generation algorithm and leave autotuning and finding the optimal loop schedule for future work.

Cycles occur in the iteration graph when an expression operates on tensors with incompatible formats, such as adding a CSR matrix to a CSC matrix, since that would require ordering the row index before the column index and vice versa. Such cycles can be resolved by inserting cycle-breaking tensor transpose operations before the actual kernel, i.e. introducing some data movement overhead in order to obtain the kernel for which efficient code can be generated. Here too, we leave finding optimal cycle resolution for future work.

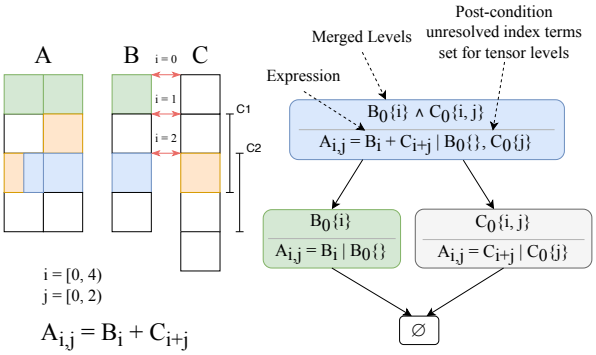
5 Code Generation

Section 5.1 introduces generalized merge lattices that handle compound subscript expressions. We then describe in detail how `subsect_next` and `subsect_begin` can be computed and generated in Section 5.2.

5.1 Generalized Merge Lattices

Merge lattices were introduced in [31] to handle the different types of merges of index variables in different iteration graphs to coiterate levels from multiple tensors. The type of merge depends on the property of the computation being generated. For multiplications, the merge is a conjunction ($\wedge$) because it requires both operands to be non-zero to produce a non-zero result (both $a \times 0 = 0$ and $0 \times b = 0$ and only $a \times b$ yields a nonzero result); for additions, the merge is a disjunction ($\vee$) since it requires only one non-zero operand to produce a non-zero result ($a + 0 = a$, $0 + b = b$, and $a + b$ yields itself). The conjunctive and disjunctive merges represent the intersection and union of sparse iteration spaces, respectively. To handle compound tensor index expressions, merges can also be composed, e.g., $(A_i + B_i) \times C_i \rightarrow (A_i \vee B_i) \wedge C_i$. The theory has been generalized [25] to handle arbitrary fill values (other than non zeros) and arbitrary semi-ring operations. For simplicity, we present our extension of merge lattices assuming non zeros to be the fill-in value and using only additions for disjunctive merges and multiplications for conjunctive merges. However, our approach can easily be adapted to the aforementioned fill values and semi-rings.

To handle the extra complexity introduced by merges of *reducing* edges, we keep a track of unresolved index terms at the current stage such that the merge lattice becomes *stateful*: A *pre-*




```
1  int pb = B.pos[0];
2  subsect ss = subsect_begin(C, 0, 2);
3
4  while (pb<B.pos[1] && !subsect_end(ss)) {
5    int ic = ss.offset;
6    int ib = B.idx[pb];
7    int vb = B.val[pb];
8    int i = min(ic, ib);
9    if (ic == i && ib == i)
10     code_for A[ic][j] = vb + ss[j]
11   else if (ib == i}
12     code_for A[ic][j] = vb
13   else if (ic == i)
14     code_for A[ic][j] = ss[j]
15
16   if (ic == i) cs = subsect_next(cs, 0, 2);
17   if (ib == i) pb++;
18 }
19 while (pb < B.pos[1]) {
20   int vb = B.val[pb], ic = B.idx[pb];
21   emit_code A[ic][j] = vb
22   pb++;
23 }
24 while (subsect_end(ss)) {
25   int ic = ss.offset;
26   emit_code A[ic][j] = cs[j]
27   ss = subsect_next(cs, 0, up(j));
28 }
```

Fig. 8. Example merge lattice for tensor expression $A_{i,j} = B_i + C_{i+j}$ and the corresponding generated code. The code segment and the corresponding node in the merge lattice are highlighted in the same color.

and *post*-condition unresolved sets are attached at each lattice point. Given a tensor addition with a compound subscript expression on C as an example, Figure 8 (left) shows how the result A is computed from two operands B and C: the value picked for $A_{i,j}$ is the result of either $B + C$, B or C based on whether both B_i and C_{i+j}, just B_i or just C_{i+j} are specified with the given i and j. The merge lattice constructed to merge index variable i is shown in figure 8 (center). In this example, to merge B_0 and C_0 on index variable i, we merge the specified index set for B_0 and the offset set for C_0 to reduce i from $i + j$, which is computed by $\mathcal{I} = \{i' | \exists j \in [0, 2)(i' \in [0, 4) \wedge (i' + j) \in \{2\})\} = \{1, 2\}$. The arrow from a lattice point to another represents that either a coordinate set or offset set is exhausted. E.g., $B_0\{i\} \wedge C_0\{i, j\} \rightarrow B_0\{i\}$ in Figure 8 (center) represents the case when there is no non-empty subsections left to be extracted from C.

Definition 5.1. A merge lattice L is a lattice that consists of n lattice points $L_1, \ldots, L_n$ and a meet operator. Associated with each lattice point L_p is a set of tensor dimensions *and its associated unresolved index terms set*, i.e., $T_p = \{\mathcal{S}_{T_0} = T_0\{i, j, \ldots\}, \mathcal{S}_{T_1} = T_1\{i, k, \ldots\}, \ldots\}$, to be merged conjunctively (i.e. $T_0 \wedge T_1 \wedge \ldots$) and an expression to be evaluated. The meet of two lattice points L_1 and L_2 with associated tensor dimensions T_1 and T_2 respectively is a lattice point with tensor dimensions $T_1 \cup T_2$. We say $L_1 \leq L_2$ if and only $T_1 \subseteq T_2$, in other words, if L_2 has tensor dimensions that are exhausted in L_1 but not vice versa.

Figure 8 (right) shows the (simplified) code corresponding to the merge lattice. Each lattice point corresponds to one `while` loop in the code, i.e., line 6-23 for $B_0\{i\} \wedge C_0\{i, j\}$, line 25-29 for $B_0\{i\}$ and line 31-35 for $C_0\{i, j\}$. Since $B_0\{i\} \wedge C_0\{i, j\}$ dominates three more lattice points (including itself), it leads to three cases inside the loop body on line 13-18. To handle the reduction of i from $C_0\{i, j\}$, we iterate over the non-empty subsection set on line 6 and line 31, with i reduced to the offset on line 7 and line 32. In this case after the reduction, there are no tensor dimension with more than one unresolved index terms, and the problem are now solvable by the existing theory (on line 14, 15, 18, 27 and 33). The reduction can be done repetitively for more complex subscript expressions.

```
code-gen(index-expr, iv) // iv is the index variable
 L = merge-lattice(index-expr, iv)
 // initialize sparse pos variables
 for Dj in sparse-dim(L):
   ivs = unresolved-ivs(Dj, L);
   if ivs.size() > 1:
     d = dim(j) // the dimension that j tied to
     // compute the subsection size.
     s = sum(iv.bound() for iv in ivs.remove(j))
     gen("auto ss{%0}{%1} = subsect_begin(D,{%2},{%3})",D,j,d,s)
   else: // falls back to TACO
     taco-gen-pos-init(...)
 for Lp in L
   // while all merged dimensions have more values
   emit "while("
   for Dj in merged-dims(Lp):
     if unreduced-ivs(Dj, L).size() > 1:
       gen("!subsect_end(ss{%0}{%1})", D, j)
     else:
       taco-gen-pos-exhuast()
   emit ") {"
   // inside loop body, initialize sparse idx variables
   for Dj in sparse-dims(Lp):
     if unresolved-ivs(Dj, L).size() > 1:
       gen("int iv{%0}{%1} = ss{%0}{%1}.offset;", D, j)
     else:
       taco-gen-sparse-iv()

   taco-gen-min-ivs()
   taco-gen-dense-pos()
   // one case per lattice point below Lp
   for Lq in sub-lattice(Lp)
     taco-gen-ivs-equals()
     for child-iv in children-in-iteraton-graph(iv)
       code-gen(expression(Lq), child-iv)
     taco-gen-compute()
   // conditionally increment the sparse pos variables
   for Dj in sparse-dims(Lp):
     if unresolved-ivs(Dj, L).size() > 1:
       gen("ss{%0}{%1} = ss{%0}{%1}.next()", D, j)
     else
       taco-increase-sparse-pos()
    gen("}")
```

(a)

```
int pb = B.pos[0];
// 2 as the upper bound of j
auto ssCi = subsect_begin(C, 0, 2);

// one loop per lattice point.
while (pb < B.pos[1] &&
       !subsect_end(ssCi)) {
  int ivCi = ssCi.offset;
  int ivBi = B.idx[pb];
  int vBi = B.val[pb];
  int i = min(ic, ib);
  // one case per lattice point.
  // C[i+j] is reduced to ssCi[j]
  if (ic == i && ib == i)
    code_for A[ic][j] = vb + ssCi[j]
  else if (ic == i)
    code_for A[ic][j] = ssCi[j]
  else if (ib == i}
    code_for A[ic][j] = vb

  // forwards to next slice.
  if (ic == i) ssCi = ssCi.next();
  if (ib == i) pb++;
}

while (pb < B.pos[1]) {
  int vb = B.val[pb], ic = B.idx[pb];
  code_for A[ic][j] = vb
  pb++;
}

while (!subsect_end(ssCi)) {
  int ic = ss.offset;
  code_for A[ic][j] = cs[j]
  ssCi = ssCi.next();
}
```

(b)

Fig. 9. (a) Recursive code generation algorithm extended from [31], only different parts are highlighted. (b) Generated code for $A_{i,j} = B_i + C_{i+j}$ (tensor A, B and C are all sparse).

Lattices can be optimized in a similar way as proposed in [31] by utilizing the fact that an index reduction loop on a dense dimension yields a dense iteration space and applies the same optimization rules.

5.2 Algorithm

Pseudo-code for the recursive code generation algorithm is given in Figure 9. Compared to the original algorithm described in [31], the new algorithm handles extra cases when the number of unresolved index variables on a tensor dimension in the current lattice point is larger than one and emits a subsection driven loop for those cases. In such cases, the extended algorithm generates:

- **`subsect_begin`** as the loop initialization condition (described in Section 5.2.1);
- **`subsect_next`** to forward the loop iteration (described in Section 5.2.2);
- **`subsect_offset`** as coordinates to co-iterate with other tensors.
- **`subsect_end`** to terminate the loop.

As described in Section 3.2, the `offset` of the subsection is memorized as metadata, thus **`subsect_offset`** be trivially computed by a load. We describe the code generation algorithm for the **`subsect_begin`** and **`subsect_next`** primitives in the following sections.

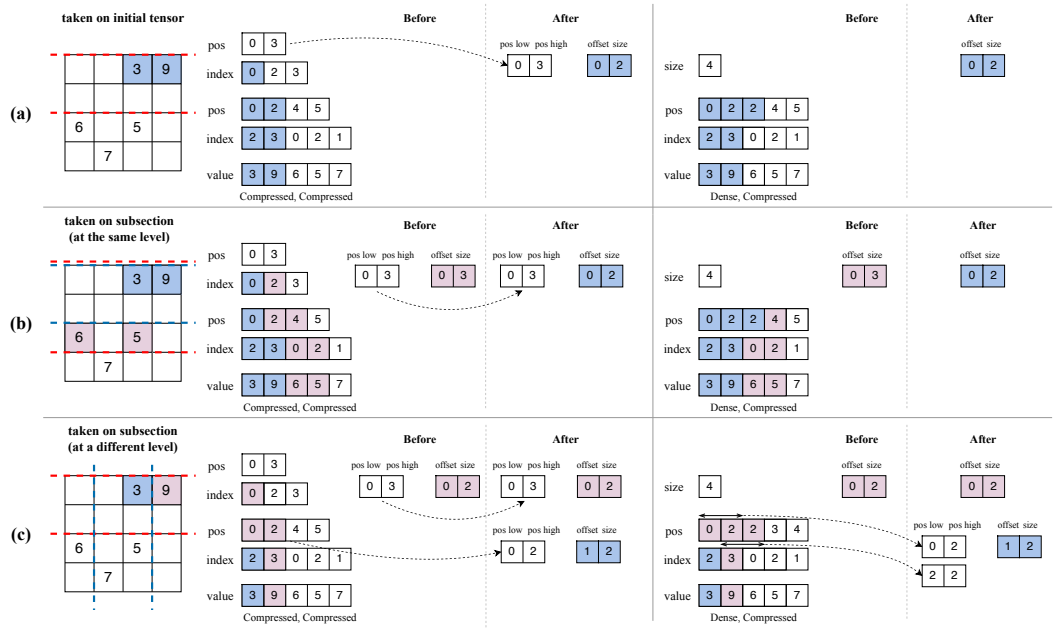


Fig. 10. Three different cases to compute `subsect_begin`. (a) computes on the initial tensor; (b) computes on a subsection of the initial tensor, both are extracted along the same dimension; (c) computes on a subsection of the initial tensor but along different dimensions. ■ marks the stored elements included in the *current* subsection. ■ marks the stored elements included in the *parent* subsection (if any).

5.2.1 Finding the first non-empty subsection. The new primitive `subsect_begin (T, D, Z)` finds the first non-empty subsection (the one with the *smallest* offset) extracted from tensor `T` along dimension `D` with size `Z`. There are three basic cases as shown in Figure 10. Note that we treat `T` as the initial tensor if it is the original tensor or extracted by *fully resolved* indices. This is because if all the index expressions on the previous dimension is fully resolved, they together locate a unique node (as opposed to a range of possible nodes) in the sparse tensor storage tree. By treating the uniquely located node as the new root of the tensor storage tree, the high-rank tensor is reduced to a lower-rank tensor and we can then compute `subsect_begin` on the rank-reduced tensor in the same way as if it is the original (unpruned) tensor.

All three cases are illustrated in Figure 10 using both CSR (with `[dense, compressed]` dimensions) and DCSR (with `[compressed, compressed]` dimensions) using the same technique introduced in Section 3.2 to specify subsections of a sparse tensor. We explain all cases in detail in the following paragraphs.

Extracting from the initial tensor: An example case is shown in Figure 10 (a). This might occur when handling tensor index expression similar to $A_{i+j,k}$ and the loop being generated is to reduce the *first* index variable from the subscript expression on the *first* dimension (say, reduce i from $i + j$ from A_0).

To handle compressed dimensions, we copy the pair of position lower and upper bound (i.e., `pos[i]` and `pos[i+1]`) from the original tensor storage as shown in Figure 10 (a) (left). We do not need to compute the position value because the first index (pointed by `index[pos[i]]`) must be included in the first subsection and, as mentioned in Section 3.2, we do not require a tight upper bound to specify the subsection (thus `pos[i+1]` can be directly reused). Assuming a sorted sparse

```
// T: the input tensor; S: context to query subsections;
// L: the extracting dimension;
// depth: the number index variables already reduced.
func genSliceBegin(Tensor T, SliceInfo S,
                   Dimension L, int depth) {
  // Generates vectors to hold the pairs.
  gen("vector<pair<int, int>> pairs_{%0}{%1};", L, depth);
  // Simple case 1: Extracting on initial tensors.
  if (L == 0 || S.resolved(L-1)) {
    if (L == 0) gen("int pos = 0;");
    else gen("int pos = {%0};", S.resolvedPos(L-1));
    // Just need to push one pair.
    gen("int pLo = T.pos_{%0}[pos];", L);
    gen("int pHi = T.pos_{%0}[pos + 1];", L);
    gen("pairs_{%0}.push_back(pLo, pHi)", L);
    return;
  }
  // Simple case 2: Extracting on the same dimension.
  if (S.preExtractDim() = L) {
    gen("pairs_{%0}{%1} = pairs_{%0}{%2}", L, depth, depth-1);
    return;
  }

  // The first level that is either resolved or sparse
  Level rootL = L - 1;
  while(rootL > 0 && !S.resolved(rootL) && dense(rootL))
    rootL --;

  if (S.resolved(rootL)) {
    gen("int pos = {%0};", S.resolvedPos(rootL));
  } else {
    if (dense(rootL)) {
      gen("for (pos = {%0}; pos < {%1}, pos++)",
          S.offset(rootL), S.bound(rootL));
    } else {
      gen("for (auto [pLo, pHi] : pairs_{%0})", rootL);
      gen("for (int pos = pLo; pos < pHi; pos++)");
      // Generate break statement.
      gen("if (T.idx_{%0}>{%1}) break;",
          rootL, S.bound(rootL));
    }
  }

  // Fills up all dense unresolved levels in between.
  for (int curL = rootL + 1; curL < L; i++) {
    gen("for (i = {%0}; i < {%1}, i++)",
        S.offset(curL), S.bound(curL));
    // Linearizes dense position.
    gen("int pos = pos * {%0} + i", T.dim(curL));
  }
  // Generates push_backs in a loop.
  gen("int pLo = T.pos_{%0}[pos];", L);
  gen("int pHi = T.pos_{%0}[pos + 1];", L);
  gen("pairs_{%d}.push_back(pLo, pHi)");
  return;
}
```

(a)

```
// T: [Compressed, Compressed]
// Dimension 0: unresolved range: [0, 2)
vector<pair<int, int>> pairs_1;
for (auto [pLo, pHi] : pairs_0) {
  for (int pos = pLo, pos < pHi; pos++) {
    if (T.idx_0[pos] > 2) break;
    pLo = T.pos_1[pos];
    pHi = T.pos_1[pos + 1];
    pairs.push_back(pLo, pHi);
  }
}
```

(b)

```
// T: [Dense, Compressed]
// Dimension 0: unresolved range: [0, 2)
vector<pair<int, int>> pairs_1;
for (int pos = 0; pos < 2; pos++) {
  pLo = T.pos_1[pos];
  pHi = T.pos_1[pos + 1];
  pairs_1.push_back(pLo, pHi);
}
```

(c)

```
// T: [Dense, Dense, Compressed]
// Dimension 0: unresolved range: [0, 2)
// Dimension 1: unresolved range: [0, 3)
vector<pair<int, int>> pairs_2;
for (int pos = 0; pos < 2; pos++) {
  for (int i = 0; i < 3; i++) {
    int pos = pos * 4 + i;
    pLo = T.pos_1[pos];
    pHi = T.pos_1[pos + 1];
    pairs_2.push_back(pLo, pHi);
  }
}
```

(d)

```
// T: [Compressed, Dense, Compressed]
// Dimension 0: resolved to pos = 1
// Dimension 1: unresolved range: [0, 3)
vector<pair<int, int>> pairs_2;
int pos = 1; /* resolved to 1 */
for (int i = 0; i < 3; i++) {
  int pos = pos * 4 + i;
  pLo = T.pos_2[pos];
  pHi = T.pos_2[pos + 1];
  pairs_2.push_back(pLo, pHi);
}
```

(e)

Fig. 11. (a) The code generation algorithm for `subsect_begin` (a) and (b) - (e) several sample code snippets generated by the algorithm.

tensor, we can load the minimum index by `index[pos[i]]`. Then, the *smallest positive* offset for the subsection that includes the minimum index is then determined by `max(0, index[pos[i]] - size + 1)`. In Figure 10 (a) (left), the offset for the first non-empty subsection is $max(0, 0-2+1) = 0$.

To handle dense dimensions, we use a sub-range determined by `[offset, offset + size)` to specify the subsection. As shown in Figure 10 (a) (right), we initialize $offset = 0, size = 2$ to record a dense sub-range.

Extracting from a subsection along the same dimension: The situation in Figure 10 (b) might occur when handling tensor index expression similar to $A_{i+j+m,k}$. For example, i is already reduced (thus a subsection has already been extracted), and the current loop being generated is to reduce j from $j + m$.

To handle compressed dimensions, we make a copy of the *entire* position array from the parent subsections. The offset for the new subsection is re-evaluated similarly by `offset = max(0, min_index - size + 1)` since the subsection's size become smaller and the previous offset might now be too small to include the minimum coordinate for the current size.

To handle dense dimensions, we adjust the value of `size` and reuse the value of `offset` from the parent subsection.

Extracting from a subsection along a different dimension: The case in Figure 10 (c) might occur when handling tensor index expression similar to $A_{i+j,k+m}$. For example, i is already reduced from $i+j$ (thus a subsection has already been extracted to reduce i), and the current loop being generated is to reduce k from $k+m$. In this case, $i+j$ and $k+m$ are subscript expressions defined on different dimensions of A.

To handle compressed dimensions, depending on whether the *previous* unresolved level is dense or compressed, different code need to be generated. In our approach, when an unresolved level is compressed, the vector of position low and high pairs must have been initialized to specify the sparse subsection. Thus, to span every pair of position low and high for the current dimension, as shown in Figure 11 (b), we iterate over previous level's pairs and for each coordinate that should be included in the subsection, we insert the position low and high pair for the current dimension. When the previous unresolved level is dense, the dense sub-range is specified by a pair of `[offset, size]`. In this case, as shown in Figure 11 (c), we iterate over `[offset, offset + size)` to populate every pair of positions ranges for the current level. At a high level, using the tensor storage tree abstraction introduced in Section 3.2, the process can be analogous to spanning all the child nodes from a set of parent nodes included in the subsection. The offset for the sparse subsection at the current dimension can be similarly determined by `offset = max(0, min_index - size + 1)`.

To handle dense dimensions, we simply use `offset = 0` together with user-provided `size` to specify the dense subsection.

Detailed Algorithm: The detailed code generation algorithm is shown in Figure 11. Line 7-9 and line 19-20 describe cases similar to Figure 11 (a) and (b) respectively. Both cases are simple and can be generated using boilerplate code. The rest of the algorithm handles general cases when there are unresolved parents dimensions. Line 29-42 generates code structures to traverse the *root* level, which is either the first level or the first non-*unresolved dense* level. Line 45-50 fills all the dense sub-ranges introduced by unresolved dense level between the root level and the current level, and line 53-55 generates the loop body to populate position ranges for the current level. The generated code is structured in a similar way shown in Figure 12.

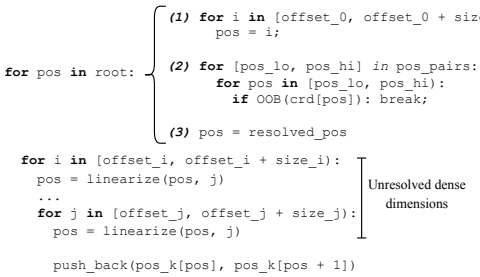


Fig. 12. The structure of generated code by Figure 11.

Figure 12 clearly shows that the size of the vector used to specified the current subsection and the time complexity to compute **subsect_begin** is proportional to $O(\prod_{i=root}^{j} size_i)$. For practical kernels like convolutions, we find that depending on the picked loop ordering, it is possible to reduce all $size_i$ to 1, leading to an $O(1)$ time complexity to compute **subsect_begin** for convolutions.

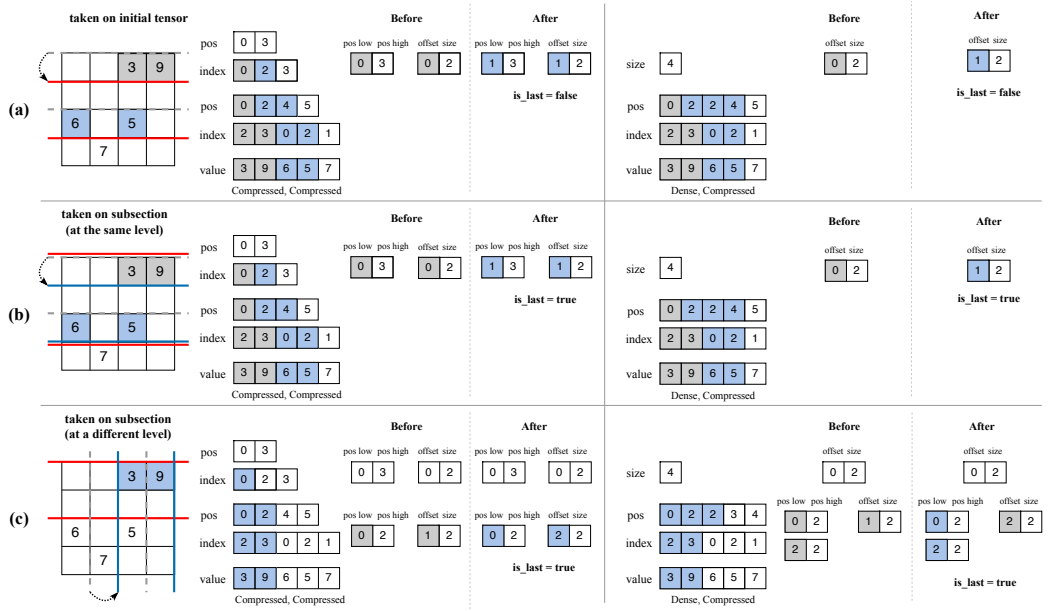


Fig. 13. Figure shows how `subsect_next` and the next non-empty slice's offset can be determined in different scenarios for each case in Figure 10. ■ marks the stored elements included in the *current* subsection. ■ marks the expired elements after forwarding the current subsection.

5.2.2 Finding the next non-empty slices. Index-reduction loops forward the offset of the current subsection in each iteration, which is computed by `subsect_next`. The process of computing the offset of the next non-empty subsection is illustrated in Figure 13. Each case in Figure 13 computes the next non-empty subsection from the state in Figure 10 respectively.

At a high level, the computation of `subsect_next` consists of two steps: ❶ expiring coordinates at the boundary (i.e., those equal to the current subsection's offset) and ❷ computing the new subsection offset from the minimum coordinates after expiration. For the simple case in Figure 13 (a) and (b) where only one pair of position range is needed to specified the subsection, the `subsect_next` is computed as below:

```
pos = offset == crd[pos] ? pos + 1 : pos;
offset = max(offset + 1, crd[pos] - size + 1)
```

For example, in Figure 13 (a), since $\texttt{index}_0\texttt{[0]} = 0$ and `0 = offset`, the position lower bound (0) increases to 1, and the minimum coordinates now becomes $\texttt{index}_0\texttt{[1]} = 2$, leading to a new non-empty subsection with `offset = 1`.

To forward an subsection that is extracted along a different dimension as the parent subsection (Fig 13 (c)), there are multiple pairs of position range. For those cases, a loop is needed to repeatedly expire coordinates and find the next smallest coordinates as there are multiple pairs of position range. The generated code repeats the same computation to forward a single position value for every expired coordinate and picks the minimal coordinates to determine the subsection offset. Same as `subsect_begin`, the time complexity to compute `subsect_next` on a sparse dimension is $O(\prod_{i=root}^{j} size_i)$.

To forward a non-empty subsection extracted from dense dimension, simply increase the offset by one.

The index-reduction loop is terminated (i.e., `subsect_end(S)` evaluated to true) if the current non-empty subsection is not fully contained by its parents, i.e., `slice.offset + slice.size > parent.size`.

5.2.3 Handling coefficients. To handle compound subscript expression with constant coefficients($c_0 \cdot i + ...b$), extra constraints are imposed on the subsections being visited. Since, by nature, induction variables in sparse iteration theory only yield positive integer value, to ensure the satisfiability of $c_0 \cdot i + ...b$, we reduce the intercept, i.e., b, by first forwarding the subsection to the minimum offset f such that $f \geq b$ and skipping subsections whose offset is not a multiple of c_0, ..., inside the loop generated to reduce i, ..., respectively.

Figure 14 shows explanatory loop structures that are generated to handle constant coefficients and/or intercept in a subscript expression(as in $c_0 \cdot i + ...b$). The constant intercept b is first reduced by forwarding the initial subsection between line 3-4 before generating any loop nest to reduce index variables. Then one loop is generated per index variable, and extra conditions are generated (line 9-10) such that only subsections with satisfied offsets are visited, e.g., checking `sec1.offset % C0 != 0` to handle c_0. Finally, i, instead of being reduced directly to `subsect.offset`, is reduced to `offset / C0` at line 11.

```
1  // Reduce b
2  sec = subsect_begin()
3  while (!sec.empty() && sec.offset < b )
4    sec = sec.next()
5
6  // Reduce C0 x i
7  sec1 = subsect_begin(sec);
8  while (!sec1.empty() {
9    while (!sec1.empty() && sec1.offset % C0 != 0)
10     sec1 = sec1.next()
11   i = sec1.offset / C0;
12   ...
13 }
```

Fig. 14. Sample code to reduce coefficients.

5.2.4 Generalization to other formats. Being able to handle arbitrary combinations of `dense` and `compressed` dimension formats (as described in Section 5.2.1 and 5.2.2) grants us the flexibility to handle a wide range of sparse formats including CSR (`[dense, compressed]`), DCSR (`[compressed, compressed]`), CSF [46] (`[dense, compressed, compressed, ...]`), etc. It is also straight forward to extend the proposed algorithm to support dimension permutations to handle, for example, column-major sparse matrices such as CSC (by constructing a column-first iteration graph [32]). The algorithm to initialize and forward subsection-driven loops can also in principle be extended to handle arbitrary dimension formats [9] as long as they are *sorted* so that the merge coiteration algorithm applies. That is, the coordinates can be retrieved in the increasing order (such that we can trivially compute the next smallest `offset` to forward the iteration).

To handle non-unique sparse dimensions (e.g., the non-unique `singleton` format used to encode COO tensors [9]), an extra "deduplication" step is required to skip duplicate coordinates. That is, instead of forwarding the next position by `pos = pos + 1`, we generate the following pseudo-code to locate the position that points to a different coordinate for non-unique dimensions:

```
while (crd[pos] == cur_coord) { pos++; }
```

6 Experimental Evaluation

To demonstrate the advantage of the technique presented in the paper, we compare our work with the most recent work, i.e., TACO-UCF on ResNet50 layers. We illustrate that our work is more general than TACO-UCF and handles more sparse formats efficiently by showing that our method is able to achieve:

- the same level of performance improvement when handling compound subscript expressions on *dense* levels in Section 6.2.1, and

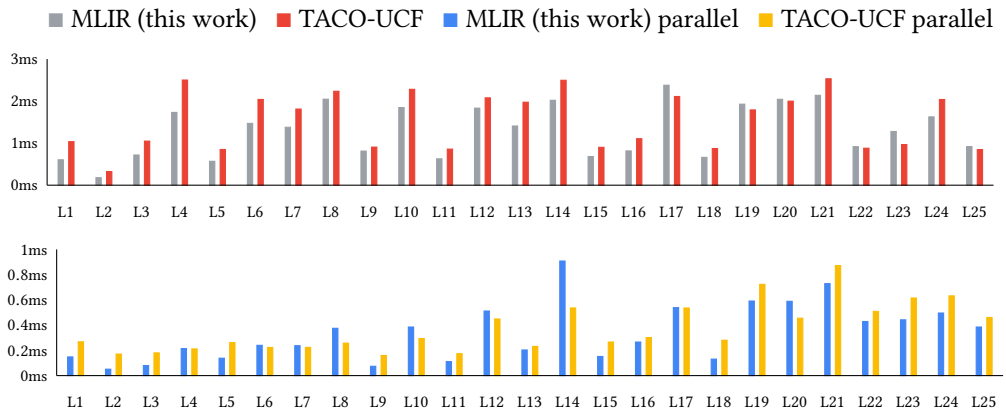


Fig. 15. Performance comparison with TACO-UCF when running on ResNet50 layers (L1-L25) with input of 80% sparsity in DDDC format.

- significant speedups when handling compound subscript expressions on *sparse* levels in Section 6.2.2.

To fully understand the potential performance benefits brought by sparsity, in Section 6.3, we also extensively evaluate our work on a set of *randomly generated* sparse tensors with sparsity ranging from 0% to 100% and compared the performance with the corresponding dense convolution kernels. In Section 6.3.1, we further analyze the results and discussed the key factors that affect the performance of the generated sparse kernels. The experiments focus solely on CPU code generation and leave GPU support as future works.

In the following sections, for sparse formats without names, we use C as the abbreviation for a compressed dimension and D as the abbreviation for a dense dimension. For example, DCC is short for a list of [Dense$_{d0}$, Compressed$_{d1}$, Compressed$_{d2}$] dimension formats to store a 3-D tensor.

6.1 Methodology

To limit the scope of the evaluation, all experiments on convolutions are conducted *with only sparse inputs*. This decision was made because we consider filter-sparse convolutions less interesting to the contribution described in the paper, that is, there is no compound subscript expressions imposed on filter tensors. Besides, filter-sparse convolution has already been supported for many years in MLIR [§].

All results are for single-threaded execution unless otherwise stated. Parallel results were obtained using 20 threads. We ran all our experiments on a Google cloudtop with a two-socket, 64-core/128-thread 2250 MHz AMD EPYC CPU that has 2 MB of L1 data cache, 32 MB of L2 cache, 256 MB of L3 cache and 512 GB of RAM. Most of the implementation of this work is open-sourced and available as a part of the MLIR sparse compiler infrastructure [3] [¶].

6.2 Comparison with TACO-UCF

6.2.1 Handling dense levels. Figure 15 shows the performance comparision between the sparse kernel generated by our algorithm and TACO-UCF (when running with both single and multiple threads) on ResNet50 [56]. Every layer of ResNet50 is a single 2-D convolution kernel with NHWC

[§] https://reviews.llvm.org/D109783

[¶] Except code related to parallelizations, which is open-sourced but has not yet been upstreamed the MLIR project by the time of writing.

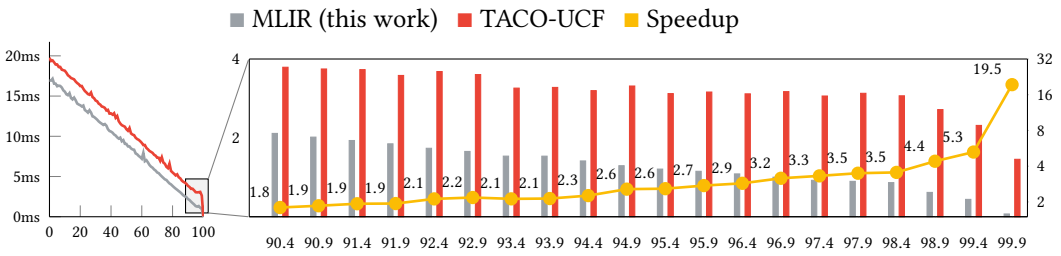


Fig. 16. Peformance comparison and speedup over TACO-UCF when running on the first layer of ResNet using DCCC format. The left part of the figure shows the performance comparison with input sparsity ranging from 0% to 100%. The right part of the figure shows the detailed performance comparison with input sparsity ranging from 90.4% to 99.9%. All input sparse tensors are randomly generated to simulate the random activation produced by the network. Speedup are plotted in log scale.

input that computes $O_{n,h,w,f} = \sum_r \sum_q \sum_c I_{n,h+r,w+q,c} F_{r,q,c,f}$ with different shapes of input and filter (when the kernel is strided, there are also extra constant coefficients on h and/or w). In this experiment, we use the exactly same input format (i.e., DDDC) and loop schedules (i.e., $n \rightarrow h \rightarrow w \rightarrow r \rightarrow q \rightarrow c \rightarrow f$) as the referenced paper [56] to evaluate our method.

On most layers, MLIR performs slightly better than TACO-UCF in both single and multi-thread settings but we also observed that MLIR performs noticeably worse at layer 14 when running in parallel. However, it is important to note that our algorithm actually generates **logically equivalent code** as TACO-UCF when inputs are in DDDC format. This is because, as indicated by the NHWC convolution formula, when inputs are in DDDC format, compound subscript expressions are only imposed on dense levels (i.e., $D_n D_{h+r} D_{w+q} C_c$). In that case, both algorithms generate dense loops to reduce an index variable. Thus, the performance differences reported in Figure 15 are mostly due to different generated target language (MLIR vs C++) and/or async runtimes (clang libomp vs gcc [‖] libomp) instead of advantages brought by the code generation algorithm. The experimental results indicate that our method provides the same level of performance boost as the state-of-the-art when handling dense levels.

6.2.2 Handling sparse levels. The advantage of our algorithm becomes more obvious when the input becomes more sparse and DDDC is no longer the optimal format to compress the tensor (multiple dense levels lead to low compress ratio). Because of a more efficient handling on compound subscript expressions imposed on *sparse* levels, the generated code by the proposed algorithm is able to fully exploit the sparsity of the input data and thus achieve significant speedup.

Figure 16 reports the execution time of generated sparse kernels by TACO-UCF and MLIR and plots the speedup of MLIR over TACO-UCF when computing the first layer of ResNet [**] using DCCC as the input format. As shown in the left part of Figure 16, MLIR consistently outperforms TACO-UCF when handling DCCC format. The performance gap also becomes more obvious as the input sparsity increases. In the right part of Figure 16 for sparsity ranging from 90.4% to 99.9%, . MLIR improves the performance by at least 1.8×. Besides, the speedup over TACO-UCF also dramatically increases as the sparsity of the input tensor increases and reaches 19.5× at 99.9% sparsity.

[‖]TACO-UCF should also have a slightly better performance when the generated code is compiled by `icc` (which has been deprecated by Intel) instead of `gcc` [56]

[**]We altered the shape of the input tensor from $1 \times 112 \times 112 \times 64$ to $1 \times 256 \times 256 \times 64$ because TACO-UCF crashed when computing DCCC inputs with the original shape.

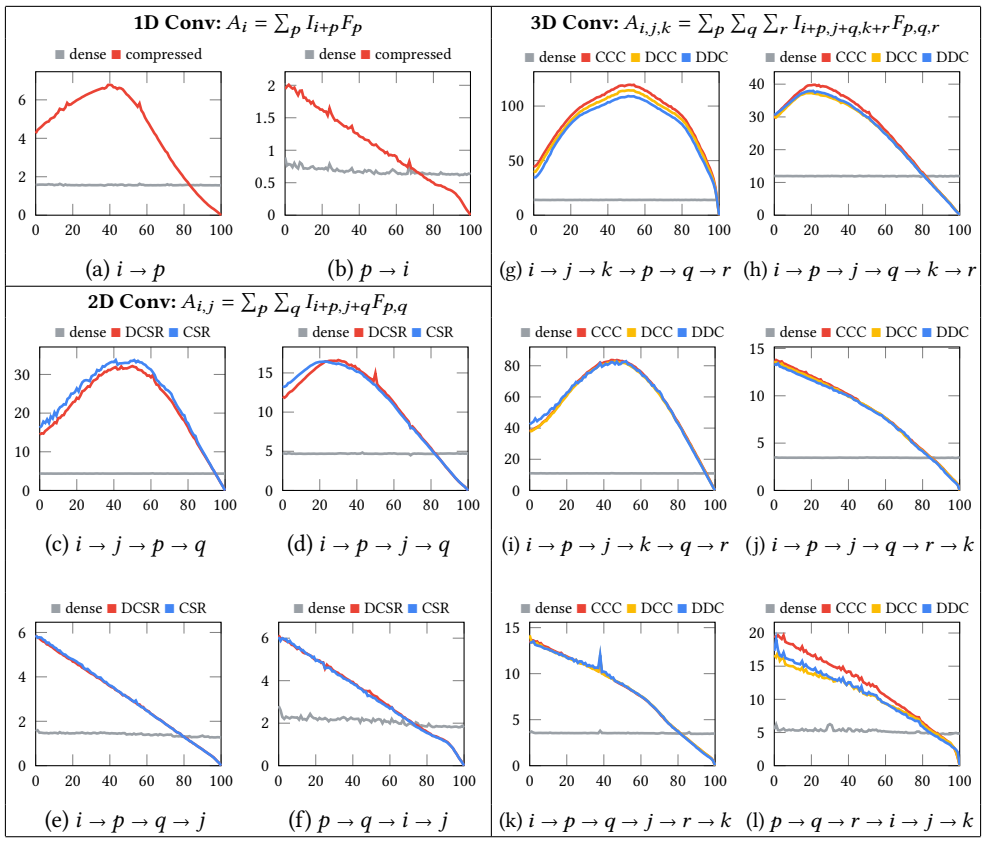


Fig. 17. Runtime (in ms) comparison between sparse and dense (1D-3D) convolutions under different loop schedules.

As shown in Figure 16, the sparse kernel generated by TACO-UCF does not show significant speedup as the input sparsity increasing until 98.4% sparsity. As the input sparsity increases from 90.4% to 98.4%, the kernel execution time of TACO-UCF only reduces from 3.8 ms to 3.1 ms (i.e., 1.2× speedup). On the other hand, the execution time of the sparse kernel generated by MLIR dramatically reduces from 2.1 ms to 0.86 ms (i.e., 2.4× speedup) for the same sparsity range. When the input becomes "hyper-sparse" (i.e., $\geq$ 98.9%), despite a noticeable performance improvement also observed in TACO-UCF generated kernels, the performance gap between MLIR only becomes larger and larger as input sparsity increases.

The reason behind the improvement is explained in Section 2: Since TACO-UCF generates a `dense` outer loop to reduce a index variable from compound subscript expressions (even for sparse levels), the benefit brought by sparsity is not fully exploited. While TACO-UCF does sparsify the inner loop, the sparse inner loop also introduce more branch instructions (to search the start position) and makes the loop bound harder to predict compared to dense loops. The irregular sparse loop hinders the instruction-level parallelism (ILP) and might shadow the performance benefit brought by sparsity, which is why the performance improvement of TACO-UCF is limited as input sparsity increases from 90.4% to 98.4%. The experiment clearly shows that our approach is more general than state-of-the-art, and can efficiently handle a much wider range of sparse formats.

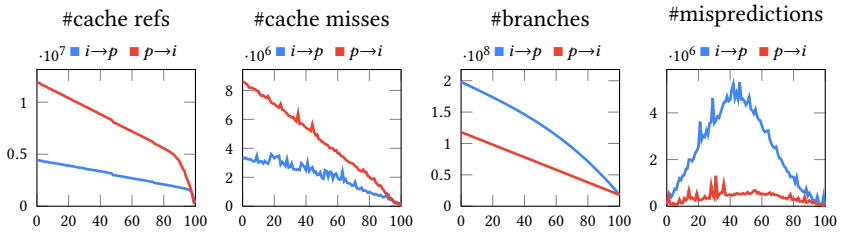


Fig. 18. Number of cache references, cache misses, executed branch instructions and branch mispredictions collected from 1D convolution with different loop schedules.

6.3 Improvement over dense kernels

To understand the performance benefit and/or penalty brought by sparsity under different formats and loop schedules, we compared the performance of the generated sparse convolution kernels against the corresponding dense kernels (compiled also by the MLIR compiler). The inputs to the convolution kernels are randomly generated in uniform distribution with sparsity ranging from 0 to 100 (with 100 being completely sparse). The input sizes for 1-D, 2-D and 3-D convolutions are 999999, 999×999 and $99 \times 99 \times 99$ respectively; the filter sizes for 1-D, 2-D and 3-D convolutions are 3, 3×3 and $3 \times 3 \times 3$ respectively (which are typical shapes of filter tensors used in practice).

Figure 17 shows the collected result. As shown in the figure, different choices of loop schedules dramatically affect the performance of the generated sparse convolution kernels. For half of the schedules (as in Figure 17 (b), (e), (f), (j), (k) and (l)), the execution time reduces *linearly* as the input sparsity increases and the sparse implementations begin to outperform the dense ones at $\sim$ 80% sparsity. For the remaining loop schedules, the generated sparse kernels are slower and start to outperform the dense ones at around 80%-98% sparsity. Despite of the impact brought by loop schedules, all generated sparse kernels, except Figure 17 (g), achieve 2×-10× speedup over dense ones at 99% sparsity regardless of *different sparse input formats*, showing the efficiency and generality of our code generation algorithm when dealing with different formats.

6.3.1 Performance analysis. Curious readers might wonder about the cause of unexpected curves reported in Figure 17 (a), (c), (d), (g), (h) and (i), in which the performance of the sparse implementation becomes worse as the sparsity of input increases from 0% to around 40%. To understand the cause of the unexpected performance degradation as inputs become sparser, we collected and plotted the number of cache accesses, cache misses, branch instructions and branch mispredictions in Figure 18 when running 1-D sparse convolutions with two different loop schedules.

Figure 18 shows that the number of cache references and executed branch instructions both decrease as the input become sparse, indicating the time complexity of the generated kernels are indeed proportional to the number of non-zeros stored in the input. Despite that schedule $i \to p$ performs worse than $p \to i$ as reported in Figure 17, it actually makes approximately 3× less cache accesses and has fewer cache misses because the intermediate result of summation to calculate $\sum_p$ is promoted to registers. On the another hand, schedule $i \to p$ results in executing more branch instructions and from the figure, the correlation between the number of misprediction and the kernel execution time (reported in Figure 17 (b)) is obvious. Since branch misprediction introduces control hazards that significantly affect instruction-level parallelism (ILP), as the number of branch mispredictions *dramatically* increases, it dominates the performance and the penalty brought by misprediction outweigh the benefit brought by sparsity, causing the performance degradation reported in Figure 17.

$i \rightarrow p$

```
1 // Subsection with the size of filter vector
2 auto sub = subsect_begin(T, 0, |p|);
3 while (!sub.empty()) {
4   int i = sub.offset;
5   while (auto [p, val] : sub) {
6     if (p >= |p|) break; ... }
7   sub = subsect_next(sub); }
```

$p \rightarrow i$

```
1 // Subsection with the size of output vector
2 auto sub = subsect_begin(T, 0, |i|);
3 while (!sub.empty()) {
4   int p = sub.offset;
5   while (auto [i, val] : sub) {
6     if (i >= |i|) break; ... }
7   sub = subsect_next(sub); }
```

Fig. 19. Pseudo-code to compute 1D convolution with different loop schedules (in our experiment $|i| = 999997$ and $|p| = 3$).

A deeper look into Figure 17 indicates that all loop schedules leading to performance degradation end with a reduction loop bounded by the shape of filter tensors, that is, loops to resolve `p, q, r` for 1-D, 2-D, 3-D convolution respectively. On the other hand, the remaining schedules all end with a parallel loop bounded by the shape of output tensors (i.e., loop `i, j, k` for 1-D, 2-D, 3-D convolution respectively). Using 1-D convolution as the example, the corresponding pseudo-code to compute Figure 17 (a) and (b) are shown in left and right of Figure 19 respectively. As described in Section 3.1, to reduce i (or p) from I_{i+p}, our algorithm constructs non-empty subsection driven loop to iterate over non-empty subsections with size p (or i) as the outer loop (L3 in Figure 19), and co-iterate the subsection with the filter vector in the inner loop (L5 in Figure 19) to resolve the remaining index variable. For 1-D convolution, since the subsection is extracted from the input tensor and there is no unresolved subscript expression on previous dimensions, **`subsect_next`** can be computed in O(1) for both schedules (as described in Section 5.2.2), thus the generated code share the same time complexity. However, extra branches to handle sparse input are introduced at L9 in Figure 19 inside **`subsect_next`** (as described in Section 5.2.2). Note that in our experiment setting where $|i| \gg |p|$, schedule $i \rightarrow p$ extracts a much smaller subsection (of size $|p|$) at L2 in Figure 19 than schedule $p \rightarrow i$, which means that compared to schedule $p \rightarrow i$, schedule $i \rightarrow p$ are more vulnerable to branch misprediction because:

- it has much more non-empty subsections to iterate over and thus executes **`subsect_next`** for much more times, which introduces more branches to predict; and
- It is harder to predict the branch at L6 in Figure 19 correctly when the subsection is smaller and there are much fewer overlapped elements between consecutive subsections.

For the generated kernel, the branching result is mostly determined by the pattern of the input data. When the input is relatively dense and there are more consecutive non-zeros in inputs, the number of misprediction is lower and thus leading to better performance. Since we generated our input completely *in random* for the experiment with no pattern, it results in a significant increase of the number of branch mispredictions, which overtakes the performance benefit brought by sparsity and leads to performance degradation. However, as we will show in Section 6.3.2, when evaluating on more structured matrices, the performance gap between different schedules can become much smaller.

6.3.2 Choice of Loop Scheduling. It might appear obvious from Figure 17 that we should always avoid an innermost loop bounded by the shape of filter matrix for convolutions to mitigate the performance penalty brought by branch misprediction. However, taking 1D convolution ($A_i = \sum_p I_{i+p} F_p$) as the example for simplicity, there are also advantages of schedule $i \rightarrow p$ over schedule $p \rightarrow i$:

- Since the output tensor is indexed by i (i.e., A_i), it is essential to have loop i as the outermost loop in order to produce results in lexicographical to directly output a sorted sparse tensor. Otherwise, a large workspace [30] need to be allocated.

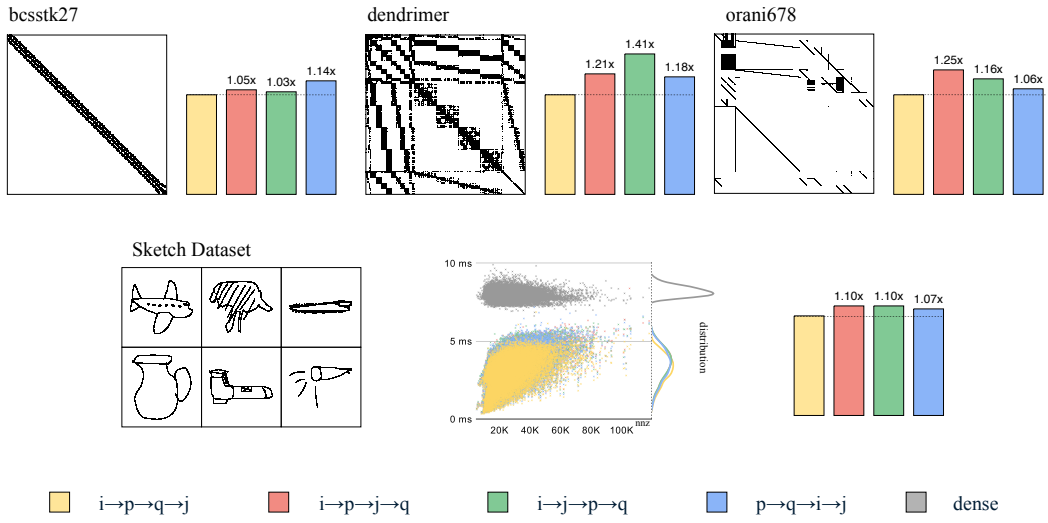


Fig. 20. Normalized execution time of the generated 2D convolution kernels (i.e., $A_{i,j} = \sum_p \sum_q I_{i+p,j+q} F_{p,q}$) with different schedules on the same matrix. The top half of the figure shows three matrices collected from SuiteSparse [11] with more predictable patterns. The bottom half of the figure shows the results when running sparse convolution on the entire sketch dataset [15] (with 20k samples) and the (normalized) average runtime for different schedules. In this experiment, the generated code produces a dense output and takes a CSR input.

- Since p is a reduction loop and i is a parallel loop, rotating loop p out of loop i introduces data dependencies on `out[i]` (L10 in Figure 19) across different iterations over p, making the outer loop hard to be parallelized and/or tiled.

Apart from the above reasons, different loop schedules might also have comparable performance when inputs have a more *predictable* sparse pattern. We picked three matrices with obvious pattern from the SuiteSparse Matrix Collection [11] and Figure 20 shows the normalized execution time of 2-D sparse convolution when running on the selected matrices. Among those picked matrices, `bcsstk27` has a typical pattern where most non-zeros appears near the diagonal; `denrimer` is a symmetrical matrix with a more complex pattern; `orani678`, while appears random, has most of its non-zeros clustered into blocks. As indicated by Figure 20, when running on `bcsstk27`, the performance gap between different schedules are almost negligible because of the simple input sparse pattern. The biggest slowdown (1.41×) is reported when running on `dendrimer`, which has $\sim$ 88% sparsity. At the same time, when running the same kernel on randomly generated inputs with the same level of sparsity, our previous experimental result reported in Figure 17 shows a $\sim$ 10× slowdown.

When running the generated convolution kernel on real-world sparse images (e.g., for sketch recognition), we found that the performance difference caused by uses of different loop orders is also small. This is because strokes in sketches are typically spatially adjacent, leading to more predictable patterns. The bottom half of Figure 20 shows the comprehensive results when running the generated sparse convolution kernels on all 20k samples in the sketch dataset [15]. As shown in the (middle) scatter plot in Figure 20, despite the chosen loop order, all generated sparse kernels are, on average, at least 2x faster than the dense implementation. While the loop order $i \rightarrow p \rightarrow q \rightarrow j$ consistently yields the best performance, the performance gaps between other loop orders are much

more negligible (compared to those collected on randomly generated sparse inputs). As shown in the bottom right of Figure 20, sub-optimal orders are at most 1.10× slower than the loop order $i \rightarrow p \rightarrow q \rightarrow j$.

The choice of loop schedules can also leads to different memory overhead introduced by forwarding a subsection as explained in Section 5.2.1. Using 2D convolution $A_{i,j} = \sum_p \sum_q I_{i+p,j+q} F_{p,q}$ as the example, schedule $p \rightarrow q \rightarrow i \rightarrow j$ requires an extra $O(i)$ buffer to maintain and forward the subsection. It is because when reducing p from $j + p$, which is the subscript expression imposed at I_1, index variable i still remains unresolved at the previous dimension I_0. On the contrary, schedule $i \rightarrow p \rightarrow j \rightarrow q$ requires only $O(1)$ extra memory since both i and p at I_0 has been reduced before handling $j + p$ at I_1.

While the impact of different loop schedules is significant for the generated sparse kernel, we focus mainly on the code generation algorithm in this paper and leave the search for the optimal schedule under different configurations as a future work.

7 Related Work

This section discussed related work, organized by traditional work on convolutions and stencil computations and compilation systems for sparse tensor algebra.

7.1 Convolutions and Stencil Computations

Convolutions form a central concept in signal and image processing as a means to translate two input signals to a, usually simpler, output signal [8]. For example, a small kernel matrix can be applied to a larger matrix in order blur an image or detect edges [47]. More recently, convolutions have become very popular in machine learning as a means of extracting features, like finding important properties in an image. In the context of deep learning, a Convolutional Neural Network (CNN) refers to a neural network that is able to learn feature engineering by itself. Early examples of these hand-written zip codes recognition [12] or phoneme recognition to discover acoustic-phonetic features[54].

Stencil computations are closely related to convolutions, since they also update tensors according to a fixed pattern, called a stencil [10]. Finding the solution to partial differential equations often results in regular or irregular stencil computations [48]. Stencils are also widely used in image processing and has been studied for decades. Frigo and Strumpen proposed a cache oblivious algorithm for stencil computation [18]. Krishnamoorthy et al. proposed an overlapping tiling strategy [34] to improve locality. By removing inter-tile dependencies, it also enables additional concurrency. The Pochoir compiler [50] transfers a serial code into a parallel one using a two-phase approach, where the programmer compiles the source program using the Pochoir template library, and then in the second phase, the programmer runs Pochoir compiler to perform a source-to-source translation. Halide [45] proposed a new domain-specific language (DSL) for the complex image processing pipelines that decouples the algorithm definition from its execution strategy to improve portability and composability, for dense array programs.

7.2 Sparsity in Neural Networks

Sparsity in neural networks is becoming a increasingly important topic. A review of sparse expert models by William Fedus, Jeff Dean and Barret Zoph concludes that "As the scale of machine learning systems has increased, the field has sought more efficient training and serving paradigms. Sparse expert models have risen as a promising solution." [17]. Many networks are also known to produce sparse activation by nature. For example, ResNet [24], due to the use of ReLU [41] layers, produces sparse activation of more than 90% sparsity [40]. Sparsity can also be introduced into the

network by pruning [23, 37] to increase model efficiency while achieve comparable accuracy at the same time [58].

7.3 Sparse Tensor Algebra Compilation

Bik and Wijshoff were the first to propose treating sparsity as a property and using a sparse compiler to generate sparse code from a sparsity-agnostic definition of the computation [4–7] and implemented the MT1 compiler to convert dense computations to the corresponding sparse ones. The Vienna-Fortran sparse extensions [52] enables programmers to characterize a matrix as sparse. The Bernoulli project [33, 39] generates efficient sparse matrix code from HPF dense loops through relational algebra to enumerate the dense iterate space as a Cartesian product. Sparsity is later introduced as predicates in filters that are then turned into relational joins, whose implementation are selected from a set of hand-coded libraries. Strout, Hall, and Olschanowsky developed the Sparse Polyhedral Framework [49], which combines polyhedral compilation with an inspector-executor approach to data inspection. The frameworks showed how sparse computation can be optimized using polyhedral techniques with uninterpreted functions to model non-affine memory accesses (e.g., accessing using coordinates loaded from a sparse tensor storage `A[crd[pos]]`) but don't currently support convolutions.

Most recently, sparse compilation was formalized and generalized to sparse tensor algebra in TACO [28–31] with a comprehensive sparse iteration theory. The MLIR sparse compiler infrastructure [3], built on top of the TACO-like iteration theory, provides an industrial-strength implementation. SparseTIR [57] further introduces an extra optimization layer in a sparse compiler and uses composable formats and transformations for performance tuning. Despite of all the recent advances in the field of sparse compilation, asymptotically optimal compiler support to handle compound subscript expressions imposed on sparse tensors remains an open problem. This paper tackles that challenge.

7.4 Sparse Convolutions

Researchers have also explored exploiting sparsity for convolutions. To exploit sparsity in the convolution filter (kernel), XNNPACK [16] implemented a `im2col`-based approach which transfers sparse convolution into SpMM. SkimCaffe [38, 43] implemented direct sparse convolutions. However, both library-based approaches share a common limitation: the algorithm is hard to generalize and only limited sparse formats are supported. The MLIR sparse compiler also supported filter-sparse convolution through a compiler-based approach [1]. We consider filter-sparse convolutions irrelevant to the contribution of this paper because there is no compound subscript expression imposed on a sparse tensor.

Aside from sparse filters, activations can also be sparse in a Convolutional Neural Network. Notably, the employment of Rectified Linear Unit (ReLU) activation function [19, 20] is known to induce sparse activations [40, 42]. SparseTrain [22] proposed a vectorzied implementation to leverage the dynamic activation sparsity introduced by ReLU by still using a dense representation for the sparse activation but avoiding unnecessary computation through vectorized zero checks. DeepSparse [35], on the other hand, adopted a "3-array" CSR-variants to compress the sparse activation.

Recently, multiple compiler-based approaches have also been proposed to handle direct convolution with sparse input. TACO-UCF [56] proposed a unified framework that is able to automatically generated code for many convolution variants, however, as shown in this paper, their approach failed to handle compound subscript expressions imposed sparse dimensions efficiently. Looplets [2],

which proposed a language to describe structural coiteration, is also able to describe a sparse convolution kernel. However, the generated code requires a binary search to locate the start of each subsection, leading to a sub-optimal performance.

7.5 Optimizing Unstructured Sparsity

As also indicated in this paper, unstructured sparsity imposes extra challenges to generate efficient sparse kernel due to high branch mispredication rate, bad cache locality, etc. Many recent works [13, 26, 55] attempt to address this issue by generating "customized" sparse kernel according to a pre-analysis on the sparsity pattern of a particular input tensor. To generate more efficient SpMM kernels with unstructured sparsity, Wilkinson et al. proposed Sparse Register Tiling [55], which uses a solver to find the best unroll-and-sparse-jam transformation. To reduce the branch misprediction rate, sparse register tiling analyzes and compresses the matrix using the provided schedule as a part of the pre-processing phase. Similarly, Horro et al. [26] proposed a custom method to generate a specialized efficiently vectorized program for the particular sparsity structure of an input matrix to compute SpMV kernels. Dezfuli and Cheshmi [13] proposed a tile fusion strategy to enable fusing GeMM-SpMM and SpMM-SpMM kernels. In their work, the scheduler also chooses the schedule based on the sparsity pattern of the input. While the mentioned systems in this section are not directly related to our work, the similar strategy (i.e., running a pre-analysis on the input sparsity pattern to guide compiler optimizations) might be adopted in the future to help generating more efficient sparse convolution kernels.

8 Conclusions and Future Research

The paper introduces a new code generation algorithm to handle tensor index expressions with compound subscript expressions on sparse tensors efficiently, which enables compiler support for complex kernels like sparse convolution. The resulting compiler generates much more efficient sparse convolution kernel than state-of-the-art compiler-based solution on CPU. The proposed method lays the foundation to generate efficient sparse convolution kernels and we believe that it enables future research to further optimizing the generated kernels including more extensive parallelization, vectorization, GPU code generation, automatic scheduling, etc.

References

[1] 2021. https://reviews.llvm.org/D109783.

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A language for structured coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 41–54.

[3] 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

[4] Aart J.C. Bik. 1996. *Compiler Support for Sparse Matrix Computations.* Ph. D. Dissertation. Department of Computer Science, Leiden University. ISBN 90-9009442-3.

[5] Aart J.C. Bik, Peter J.H. Brinkhaus, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. 1998. The Automatic Generation of Sparse Primitives. *Transactions on Mathematical Software* 24 (1998), 190–225.

[6] Aart J.C. Bik and Harry A.G. Wijshoff. 1995. Advanced Compiler Optimizations for Sparse Computations. *J. Parallel and Distrib. Comput.* 31 (1995), 14–24.

[7] Aart J.C. Bik and Harry A.G. Wijshoff. 1996. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Transactions on Parallel and Distributed Systems* 7, 2 (1996), 109–126.

[8] Giovanni Capobianco, Carmine Cerrone, Andrea Di Placido, Daniel Durand, Luigi Pavone, Davide Russo, and Fabio Sebastiano. 2021. Image convolution: a linear programming approach for filters design. *Soft Computing* 25 (07 2021). https://doi.org/10.1007/s00500-021-05783-5

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

[10] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas) *(SC '08)*. IEEE Press, Article 4, 12 pages.

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

[12] John Denker, W. Gardner, Hans Graf, Donnie Henderson, R. Howard, W. Hubbard, L. D. Jackel, Henry Baird, and Isabelle Guyon. 1988. Neural Network Recognizer for Hand-Written Zip Code Digits. In *Advances in Neural Information Processing Systems*, D. Touretzky (Ed.), Vol. 1. Morgan-Kaufmann. https://proceedings.neurips.cc/paper_files/paper/1988/file/a97da629b098b75c294dffdc3e463904-Paper.pdf

[13] Mohammad Mahdi Salehi Dezfuli and Kazem Cheshmi. 2024. Improving Locality in Sparse and Dense Matrix Multiplications. *arXiv preprint arXiv:2407.00243* (2024).

[14] Iain S. Duff, A.M. Erisman, and J.K. Reid. 1990. *Direct Methods for Sparse Matrices*. Oxford Science Publications, Oxford.

[15] Mathias Eitz, James Hays, and Marc Alexa. 2012. How Do Humans Sketch Objects? *ACM Trans. Graph. (Proc. SIGGRAPH)* 31, 4 (2012), 44:1–44:10.

[16] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. 2019. Fast Sparse ConvNets. *CoRR* abs/1911.09723 (2019). arXiv:1911.09723 http://arxiv.org/abs/1911.09723

[17] William Fedus, Jeff Dean, and Barret Zoph. 2022. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667* (2022).

[18] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*. 361–366.

[19] Kunihiko Fukushima. 1969. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics* 5, 4 (1969), 322–333.

[20] Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics* 36, 4 (1980), 193–202.

[21] Alan George and Joseph W.H. Liu. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliffs, New York.

[22] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, and Josep Torrellas. 2020. SparseTrain: Leveraging dynamic sparsity in software for training DNNs on general-purpose SIMD processors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 279–292.

[23] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[25] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[26] Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. 2022. Custom High-Performance Vector Code Generation for Data-Specific Sparse Computations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 160–171.

[27] Kevin Hunter, Lawrence Spracklen, and Subutai Ahmad. 2022. Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks. *Neuromorphic Computing and Engineering* 2, 3 (jul 2022), 034004. https://doi.org/10.1088/2634-4386/ac7c8a

[28] Fredrik Kjolstad. 2020. *Sparse Tensor Algebra Compilation*. Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[29] Fredrik Kjolstad et al. 2017. TACO: The Tensor Algebra Compiler. Open-source project available at http://tensor-compiler.org/.

[30] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (2019), 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

[31] 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

[32] Fredrik Berg Kjølstad. 2020. *Sparse tensor algebra compilation*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[33] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *European Conference on Parallel Processing*. Springer, 318–327.

[34] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2007. Effective automatic parallelization of stencil computations. *ACM sigplan notices* 42, 6 (2007), 235–244.

[35] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Bill Nell, Nir Shavit, and Dan Alistarh. 2020. Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, Virtual, 5533–5543. http://proceedings.mlr.press/v119/kurtz20a.html

[36] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). arXiv:2002.11054 https://arxiv.org/abs/2002.11054

[37] Joo Hyung Lee, Wonpyo Park, Nicole Elyse Mitchell, Jonathan Pilault, Johan Samir Obando Ceron, Han-Byul Kim, Namhoon Lee, Elias Frantar, Yun Long, Amir Yazdanbakhsh, et al. 2024. JaxPruner: A concise library for sparsity research. In *Conference on Parsimony and Learning*. PMLR, 515–528.

[38] Sheng R. Li, Jongsoo Park, and Ping Tak Peter Tang. 2017. Enabling Sparse Winograd Convolution by Native Pruning. *CoRR* abs/1702.08597 (2017). arXiv:1702.08597 http://arxiv.org/abs/1702.08597

[39] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. 2000. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 14th international conference on Supercomputing*. 88–99.

[40] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. arXiv:2310.04564 [cs.LG]

[41] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[42] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 907–922.

[43] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2017. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJPcZ3txx

[44] Sergio Pissanetsky. 1984. *Sparse Matrix Technology*. Academic Press, London.

[45] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 1–12.

[46] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 1–7. https://doi.org/10.1145/2833179.2833183

[47] Haldo Spontón and Juan Cardelino. 2015. A Review of Classic Edge Detectors. *Image Processing On Line* 5 (2015), 90–123. https://doi.org/10.5201/ipol.2015.35.

[48] Michelle Mills Strout. 2013. Compilers for Regular and Irregular Stencils: Some Shared Problems and Solutions. In *Proceedings of Workshop on Optimizing Stencil Computations (WOSC)*.

[49] 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.

[50] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 117–128.

[51] Reginal P. Tewarson. 1973. *Sparse Matrices*. Academic Press, New York, NY.

[52] Manuel Ujaldon, Emilio L. Zapata, Barbara M. Chapman, and Hans P. Zima. 1997. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems* 8, 10 (1997), 1068–1083.

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

[54] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. 1989. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, 3 (1989), 328–339. https://doi.org/10.1109/29.21701

[55] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1995–2020.
[56] 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).
[57] 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*. 660–678.
[58] Yihua Zhang, Yuguang Yao, Parikshit Ram, Pu Zhao, Tianlong Chen, Mingyi Hong, Yanzhi Wang, and Sijia Liu. 2022. Advancing model pruning via bi-level optimization. *Advances in Neural Information Processing Systems* 35 (2022), 18309–18326.
[59] Zahari Zlatev. 1991. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, Dordrecht.

Received 2024-04-04; accepted 2024-08-18