UniSparse: An Intermediate Language for General Sparse Format Customization

JIE LIU, Cornell University, USA
ZHONGYUAN ZHAO, Cornell University, USA
ZIJIAN DING, University of California, Los Angeles, USA
BENJAMIN BROCK, Intel, USA
HONGBO RONG, Intel, USA
ZHIRU ZHANG, Cornell University, USA

The ongoing trend of hardware specialization has led to a growing use of custom data formats when processing sparse workloads, which are typically memory-bound. These formats facilitate optimized software/hardware implementations by utilizing sparsity pattern- or target-aware data structures and layouts to enhance memory access latency and bandwidth utilization. However, existing sparse tensor programming models and compilers offer little or no support for productively customizing the sparse formats. Additionally, because these frameworks represent formats using a limited set of per-dimension attributes, they lack the flexibility to accommodate numerous new variations of custom sparse data structures and layouts.

To overcome this deficiency, we propose UniSparse, an intermediate language that provides a unified abstraction for representing and customizing sparse formats. Unlike the existing attribute-based frameworks, UniSparse decouples the logical representation of the sparse tensor (i.e., the data structure) from its low-level memory layout, enabling the customization of both. As a result, a rich set of format customizations can be succinctly expressed in a small set of well-defined query, mutation, and layout primitives. We also develop a compiler leveraging the MLIR infrastructure, which supports adaptive customization of formats, and automatic code generation of format conversion and compute operations for heterogeneous architectures. We demonstrate the efficacy of our approach through experiments running commonly-used sparse linear algebra operations with specialized formats on multiple different hardware targets, including an Intel CPU, an NVIDIA GPU, an AMD Xilinx FPGA, and a simulated processing-in-memory (PIM) device.

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

Additional Key Words and Phrases: sparse data formats, compilers, programming languages, heterogeneous systems

ACM Reference Format:
Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. 2024. UniSparse: An Intermediate Language for General Sparse Format Customization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 99 (April 2024), 29 pages. https://doi.org/10.1145/3649816

Authors' addresses: Jie Liu, Cornell University, Ithaca, USA, jl3952@cornell.edu; Zhongyuan Zhao, Cornell University, Ithaca, USA, zhozh@qti.qualcomm.com; Zijian Ding, University of California, Los Angeles, Los Angeles, USA, bradyd@cs.ucla.edu; Benjamin Brock, Intel, San Jose, USA, benjamin.brock@intel.com; Hongbo Rong, Intel, San Jose, USA, hongbo.rong@intel.com; Zhiru Zhang, Cornell University, Ithaca, USA, zhiruz@cornell.edu.

ACM 2475-1421/2024/4-ART99
https://doi.org/10.1145/3649816

1 INTRODUCTION

As Dennard scaling ended in the mid-2000s and Moore's Law is approaching its limit, computer engineers are increasingly turning to special-purpose hardware accelerators to meet the ever-growing computational demands of emerging application domains such as graph analytics, machine learning, and robotics. At the same time, there has been an explosion in the amount of data that domain experts have to manage. Notably, much of this big data is sparse in nature. For example, Amazon co-purchase graphs have 400K nodes and a density of 0.002%, and arXiv graph datasets have 100M papers and a density of 0.00002% [Hu et al. 2020; Leskovec and Krevl 2014]. These evident trends in technology and applications are driving computing systems towards heterogeneity that can process sparse data in an efficient and high-performance manner.

Many important operations (i.e., kernels) of sparse processing are performed on sparse tensors, a generalization of sparse matrices. Sparse tensors are typically represented using specialized *data structures* that leverage the sparsity of the tensor to reduce storage size and/or memory footprint. These data structures usually store only the non-zero elements (or non-zero blocks) of the tensor, along with their associated coordinates that are encoded in a compressed form as *metadata*. Various forms of *data layouts*, such as the structure of arrays (SoA) or array of structures (AoS), can be employed to store the sparse data structure in memory. The data structure and data layout jointly determine a *sparse format*.

The metadata can be viewed as a hierarchical tree that captures the multi-dimensional coordinates of the non-zeros in a structured way. In this work, we refer to this tree as the *metadata tree* and use it as a logical representation of the sparse format. To reconstruct the original coordinates of a non-zero element, multiple indirect memory accesses are required to traverse the metadata tree. Due to the input-dependent and irregular data access patterns that result from this process, sparse workloads are typically memory-bound.

To efficiently utilize memory bandwidth, reduce memory accesses, and exploit data parallelism to boost the performance of sparse tensor computation, researchers are increasingly using custom sparse formats optimized for particular application domains and/or target hardware architectures. Examples include hybrid formats for GPUs [Bell and Garland 2009; Choi et al. 2010; Guo et al. 2016] and banked formats for FPGAs [Hu et al. 2021; Fowers et al. 2014] and dedicated accelerators [Srivastava et al. 2020]. While format customization can significantly improve performance, we recognize two pressing issues: i) *productivity* – it takes substantial engineering effort to design a custom sparse format and adapt the implementation of related compute operations that must interact with the new format, and ii) *permutability* – there lacks a unified abstraction that can systematically encode different variants (or permutations) of existing sparse formats to facilitate the exploration of a complex design space, where the search of custom formats needs to account for non-zero distribution patterns of inputs, inherent parallelism of the dominant compute kernels, and the target hardware.

Prior research has attempted to address the productivity challenge by using either manually optimized libraries or automatic compilers. Sparse linear/tensor algebra libraries (e.g., sparse BLAS, Intel MKL, NVIDIA cuSPARSE) provide highly optimized target-specific sparse kernels. While library functions achieve high performance, they only support a limited set of sparse formats. Recent efforts on sparse tensor algebra compilers such as TACO [Kjolstad et al. 2017; Chou et al. 2018], COMET [Tian et al. 2021], and SparseTIR [Ye et al. 2023] describe tensor dimensions in attributes (e.g., `dense` or `compressed`), and generate sparse tensor algebra kernels assisted by predefined code generation templates. This attribute-based format abstraction limits their extensibility to support new custom formats, as the finite combinations of attributes restrict the range of possible data structures, and the fixed code generation templates restrict the possible data layouts. As a result,

this abstraction offers programmers no further customization opportunities for the data structures and layouts, and important details about the data structures and layouts necessary for identifying a specific format uniquely may be omitted.

This work proposes *UniSparse*, which is the first intermediate language designed for general sparse format customization. UniSparse aims to (1) provide a systematic way of expressing an unlimited number of custom formats, (2) support format customization at both the logical data structure and physical layout level, while taking into account the sparsity patterns of input tensors, compute operations, and hardware targets, and (3) automate code generation for compute operations and conversion with other formats for the newly defined formats. With UniSparse, the metadata tree serves as a logical representation that can be expressed using an *index map*, a set of *query* and *structural mutation* operations, which we call *primitives* (§4.1). An additional set of *layout* primitives specifies how to partition and traverse the metadata tree, which transforms the tree into physical memory layouts (§4.2). The index map and primitives are the essential components of a succinct *intermediate language* for specifying custom sparse formats, including but not limited to many previously proposed high-performance formats.

Compared to the previous attribute-based approach, the UniSparse language offers a holistic representation of sparse formats using mapping functions and primitives, without presuming dimension-wise composition of separate data structures. The language decouples logical representations of sparse formats from their physical memory layouts, allowing both to be specialized. Empowered by the language, the UniSparse compiler can reason formally about the correspondence between various formats and their underlying layouts, automating both code generation and format conversion. To this end, we develop a data structure and data layout inference algorithm (§5.1) that automatically determines the storage format of sparse tensors. Furthermore, we introduce a format conversion algorithm (§5.2) that enables the compiler to handle a broad range of source and destination formats, including both conventional and specialized ones. We also provide a general compute kernel generation algorithm (§5.3) that supports custom sparse formats.

The UniSparse compiler is built on top of the MLIR infrastructure [Lattner et al. 2020] (§6). We include the compiler as an artifact for evaluation, and in Section 7, we demonstrate its efficacy in customizing formats on multiple hardware platforms, including CPUs, GPUs, FPGAs, and a simulated PIM device [Devic et al. 2022]. It can also automate the conversion among a variety of sparse formats, resulting in significant productivity improvements.

2 BACKGROUND AND MOTIVATION

This section overviews common sparse formats (Figure 1), and prior research and their limitations to motivate our work (§2.2). Sparse matrices are used as illustrative examples for simplicity, while the discussion generalizes to tensors.

2.1 Sparse Formats

To improve performance and adapt to various architectures and sparsity patterns, numerous tensor formats have been developed.

The coordinate (COO) format [Bader and Kolda 2008] (Figure 1b) stores non-zero values along with their complete coordinate information. COO is widely used as the default format for many sparse data files, such as the Matrix Market (.mtx) [Boisvert et al. 1996] and FROSTT (.tns) [Smith et al. 2017] file formats. While COO stores row and column indices as separate arrays, the dictionary of keys (DOK) format [Johansson and Johansson 2015] (Figure 1f) adopts a different memory layout by pairing up indices in a single array. Sparse tensors may have multiple non-zero values per row, and many non-zero elements can share a common row index. The compressed sparse row (CSR) format (Figure 1c) replaces these row coordinates with a compact pointer array that slices values belonging

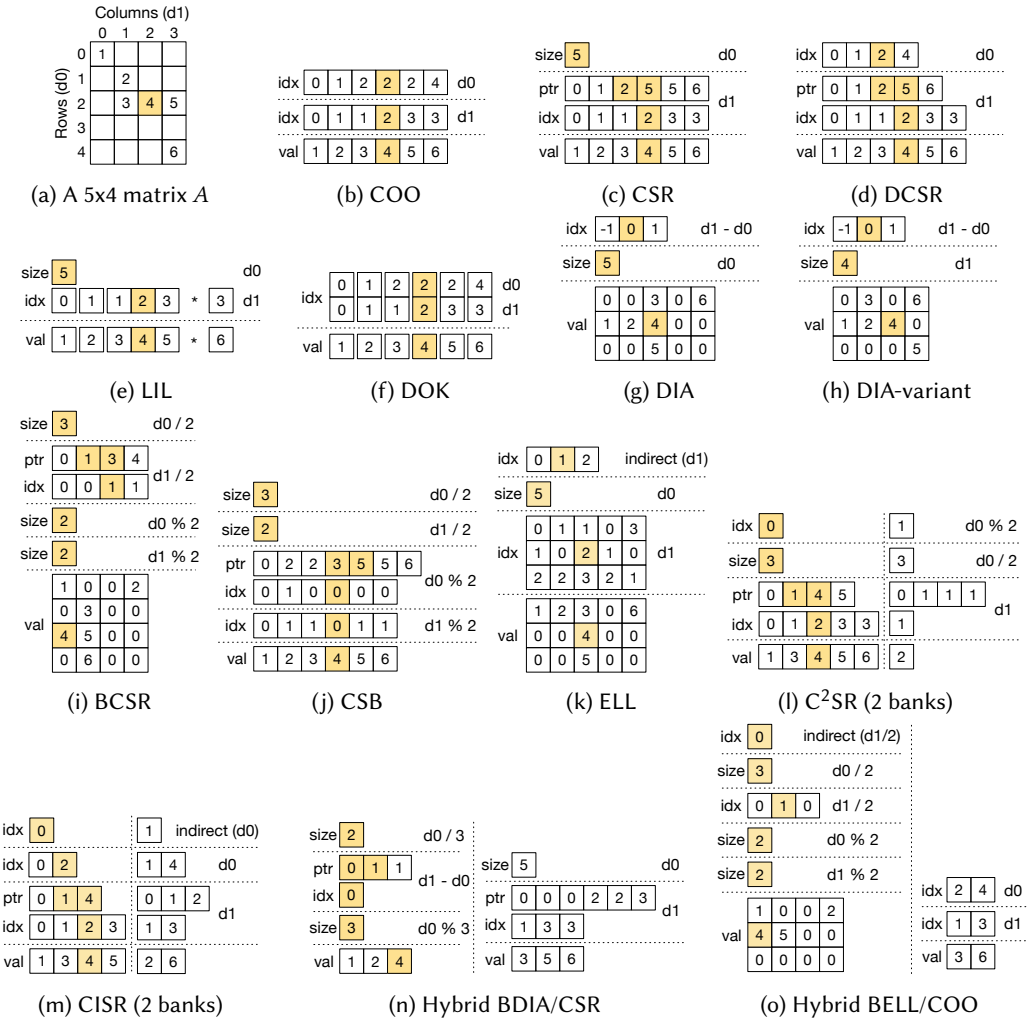


Fig. 1. Different formats of a sparse matrix (A) — Yellow-shaded blocks refer to the data structure of the same tensor element. The *size*, *ptr*, *idx*, and *val* labels on the left indicate the size of a dimension, a pointer array, an index array, or a value array. The index expressions on the right denote the index map at each level. Horizontal dotted lines separate different tensor dimensions, while vertical dotted lines divide the matrix A into sub-matrices.

to different rows, thus saving more space than COO. The linked list (LIL) format [Johansson and Johansson 2015] (Figure 1e) employs an alternative physical layout of the CSR format, where column indices and values are stored separately per row, eliminating the need for row pointers. The doubly-compressed sparse row (DCSR) format [Buluc and Gilbert 2008] (Figure 1d) provides further optimizations for matrices with many empty rows by storing the pointers array as a compressed list and avoiding storing pointers for empty rows in the pointers array.

To efficiently store matrices with non-zero values clustered along the diagonals, the diagonal (DIA) format [Saad 2003] (Figure 1g) stores only the diagonals containing non-zero values. The traditional DIA format pads diagonals to the full size of the row dimension (`d0` in Figure 1), while a

variant of the DIA format (Figure 1h) pads diagonals to the full size of the column dimension (d1), which is more space-efficient for matrices with fewer columns.

Blocked formats partition a tensor into sub-tensor chunks. For instance, a blocked variant of CSR, known as the block compressed sparse row (BCSR) format [Im and Yelick 1998] (Figure 1i), stores a compressed collection of small dense matrix blocks. In contrast, the compressed sparse block (CSB) format [Buluç et al. 2009] (Figure 1j) stores a dense collection of matrix blocks in a compressed format. BCSR improves register reuse, while CSB exposes parallel execution opportunities in sparse matrix-vector multiplication (SpMV). Another format that balances workloads for vectorized processing is the ELLPACK (ELL) format [Kincaid et al. 1989] (Figure 1k). It packs non-zero values in each row and reduces the column size to the maximum amount of non-zero values per row.

Hardware accelerators for sparse processing can benefit from formats customized per data access patterns or memory systems. One such format is the cyclic channel sparse row (C^2SR) format [Srivastava et al. 2020] (Figure 1l). This format partitions a tensor's rows into sub-tensors, one sub-tensor for each memory bank, increasing memory bandwidth utilization. Another example is the compressed interleaved sparse row (CISR) format [Fowers et al. 2014] (Figure 1m), which schedules rows of the CSR format to different compute units of an accelerator, balancing the workload and eliminating memory access conflicts.

Formats can also be customized based on the sparsity patterns of input tensors. A hybrid format represents sub-tensors with their best suitable formats independently. One example is the hybrid blocked-DIA (BDIA) format [Fukaya et al. 2021] and CSR format (Figure 1n), which achieves higher performance for datasets with non-zeros clustered along diagonals on multi-thread CPUs by increasing data temporal locality in cache blocks. Another example is the hybrid blocked-ELLPACK (BELL)/COO format (Figure 1o), which combines several format optimization techniques [Guo et al. 2016; Bell and Garland 2009; Choi et al. 2010].

2.2 Prior Work on Sparse Format Abstraction

Early sparse tensor algebra compilers [Bik and Wijshoff 1993; Pugh and Shpeisman 1999] transform dense linear algebra programs to runnable sparse code with sparsity predicates (guards), but only a few formats are supported with hard-coded format descriptions. The idea of supporting different data structures with a format abstraction was pioneered by the Bernoulli Compiler [Kotlyar et al. 1997], which introduces an index hierarchy and per-dimension accessing rules to describe various sparse formats. Another work [Arnold et al. 2010] defines a functional language for specifying sparse matrix formats as a sequence of constructs that facilitates code verification.

Recent years have seen a surge of research on sparse tensor compilers. TACO [Kjolstad et al. 2017; Chou et al. 2018] represents sparse formats as per-dimension attributes and generates code for tensor algebra expressions based on the attributes. Automated format conversion is proposed in [Chou et al. 2020a], which uses index maps and queries to specify formats in a more programmable way. The MLIR SparseTensor dialect [Bik et al. 2022] is a recent effort that leverages TACO's code generation theories to build a sparse linear algebra compiler within the MLIR infrastructure. Other compiler frameworks, such as COMET [Tian et al. 2021], also use attribute-based format abstractions similar to TACO and target heterogeneous architectures and computational chemistry workloads. Additionally, SparseTIR [Ye et al. 2023] generates high-performance sparse kernels for machine learning workloads on GPUs using hybrid formats. Table 1 summarizes recent work on sparse tensor algebra compilers with format abstraction, categorized into two classes:

Attribute-Based Format Abstraction. Previous research, such as TACO [Kjolstad et al. 2017; Chou et al. 2018], MLIR's SparseTensor dialect [Bik et al. 2022], COMET [Tian et al. 2021], and SparseTIR [Ye et al. 2023], uses per-dimension attributes to describe formats. For example, in TACO, the CSR format is encoded as `{dense, compressed}` in row and column dimensions, respectively.

Table 1. State-of-the-art sparse tensor algebra compilers.

Abstraction	Prior Work	Custom Index Maps	Sparsity Pattern-Aware Hybrid Formats	Backend-Aware Memory Layouts	Automated Conversion
Attribute-based	TACO [Kjolstad et al. 2017], [Chou et al. 2018]	○	○	◓	○
	MLIR SparseTensor [Bik et al. 2022]	◓	○	◓	◓
	COMET [Tian et al. 2021]	○	○	◓	○
	SparseTIR [Ye et al. 2023]	◓	◓	◓	○
Language-based	LL [Arnold et al. 2010]	◓	○	◓	○
	TACO-conversion [Chou et al. 2020a]	◓	○	◓	●
	UniSparse (This Work)	●	●	●	●

Similarly, SparseTIR uses `{I = dense_fixed(I_Size, dataType), J = sparse_variable(I, ..., dataType)}` to represent the same format.

However, limited by the finite combinations of attributes, the attribute-based approach is unable to support a vast number of new formats. For instance, while TACO can express the traditional DIA format (Figure 1g), it cannot accommodate the DIA-variant format (Figure 1h). Additionally, TACO lacks support for custom index maps, which are necessary for the CISR format (Figure 1m). MLIR's SparseTensor dialect and SparseTIR have incorporated index expressions, which enhance expressiveness, but do not entirely overcome the limitation. Moreover, the prior approach generates compute operations with fixed iteration and access templates, resulting in non-customizable memory layouts. Additionally, different memory layouts such as SoA and AoS are not distinguished. Lastly, since attributes do not directly reflect the memory layouts, implementing fully automated format conversion becomes challenging.

Language-Based Format Abstraction. Earlier research [Arnold et al. 2010] defines a sparse format by specifying the compression process using a functional language. However, this approach strongly couples the specification of compute operations with the format description, leading to significant variations in user programs for the same compute kernel with different formats. Another work [Chou et al. 2020a] introduces a language to assist format conversion. It supports index map functions and blocking formats, and thus can handle formats such as the DIA-variant (Figure 1h), BCSR (Figure 1i), and CSB (Figure 1j). However, it still does not support custom index maps required by CISR (Figure 1m), or discernible layouts required by LIL (Figure 1e) and DOK (Figure 1f). Moreover, sparsity pattern-aware hybrid formats are not supported.

To the best of our knowledge, UniSparse is the first language for format abstraction that can encode a wide range of custom formats including those with custom index maps, hybrid formats aware of sparsity patterns, and target-specific memory layouts.

3 OVERVIEW OF UNISPARSE

UniSparse is an intermediate language designed to formally and succinctly represent a wide range of sparse formats, along with a compiler that automates both format conversion and customization. Figure 2 offers an overview of UniSparse's design, with the format abstraction forming the foundation of the entire framework. We develop an intermediate language that enables users to

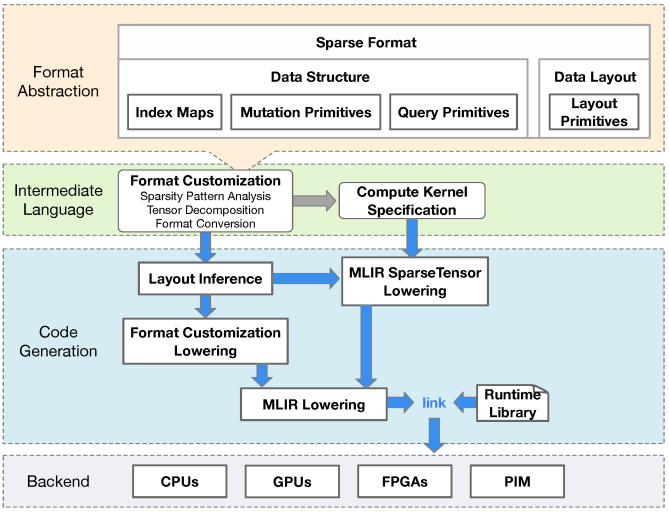


Fig. 2. An overview of UniSparse.

specify both format customization and compute kernels independently. The intermediate language program can run on various backends through automated code generation.

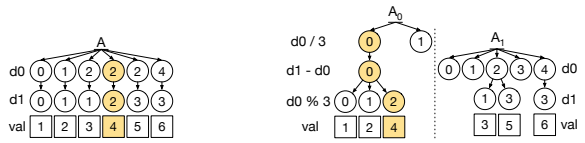
Our approach to sparse tensor format abstraction revolves around a succinct yet expressive representation for both data structures and layouts. The data structures of a format are logically represented as a metadata tree (§4.1), encoded by an index map (§4.1.1) and a set of query (§4.1.2) and mutation primitives (§4.1.3). From the metadata tree, we obtain the data layouts of a format using layout primitives (§4.2).

During the code generation stage (§5), the UniSparse compiler first infers the data structures and layouts from format encodings (§5.1). Once the formats are determined, the compiler proceeds to lower format customization and compute operations. For format conversion, the compiler automatically applies a sequence of rewrite rules to convert from the source format to the destination format (§5.2). To lower compute operations, UniSparse leverages the MLIR SparseTensor dialect for its supported conventional formats. The compute kernels for custom formats are generated based on the proposed two-step algorithm (§5.3). The UniSparse intermediate language (§6) is implemented as a new dialect in MLIR. Our compiler can support multiple hardware targets, including CPUs, GPUs, FPGAs, and a PIM simulator [Devic et al. 2022].

As shown in Figure 2, UniSparse follows a design principle of decoupling format abstraction and compiler implementation at three levels:

- Data structures and memory layouts are expressed in a decoupled way, enabling support for a broader range of custom sparse formats.
- The format customization and compute operations are specified independently, freeing developers from tedious format-specific implementation details when writing compute kernels.
- The format abstraction is independent of the input language, making it portable and complementing prior attribute-based approaches.

An Illustrative Example. Figure 3 illustrates our approach using the SpMV kernel with an input matrix (Figure 1a), originally in the COO format (Figure 3a) and then converted to the hybrid BDIA/CSR format (Figure 3b). Although we will cover the formal syntax and technical details of the UniSparse language in later sections, the purpose of this example is to provide an intuitive understanding of how we address the following questions: (1) The sparsity patterns of the matrices are input-dependent. How to decompose the matrix according to the non-zero distribution patterns?




```
// COO to BDIA
COO_structure
.Skew(0, 1, -1)
.TileSplit(0, 3).Sort()
.Fill(0).Merge(0)
// COO to CSR
COO_structure
.Fill(0).Merge(0)
```

(a) Source data structure: COO (b) Target data structure: BDIA/CSR (c) Convert COO to BDIA/CSR.

```
1  // Format abstraction
2  #COO = #unisparse.encoding<{
3          idx_map = #unisparse.map<(d0,d1)->(d0,d1)>, mutation = #unisparse.prim<trim(0,1)> }>
4  #CSR = #unisparse.encoding<{
5          idx_map = #unisparse.map<(d0,d1)->(d0,d1)>, mutation = #unisparse.prim<merge(0), trim(1,1)> }>
6  #BDIA = #unisparse.encoding<{
7          idx_map = #unisparse.map<(d0,d1)->(d0/3,d1-d0,d0%3 )>, mutation = #unisparse.prim<merge(0),
       trim(1,1)> }>
8  #COO_COO = #unisparse.hybrid<{ fmats = [#COO, #COO] }>
9  #BDIA_CSR = #unisparse.hybrid<{ fmats = [#BDIA, #CSR] }>
10 // Format pre-processing
11 #sum = #unisparse.sum< groupBy (d0,d1) -> (d0/3,d1-d0), with val ne 0 -> 1 | otherwise -> 0 >
12 %A1 = unisparse.decompose (%in_A, %thld) { query = #sum }: tensor<?x?xf32, #COO_COO>
13 %A2 = unisparse.convert (%A1): tensor<?x?xf32, #BDIA_CSR>
14 // Compute operation
15 #spmv = { indexing_maps = [
16   affine_map<(d0,d1)->(d0,d1)>, // for argument %A2
17   affine_map<(d0,d1)->(d1)>,     // for argument %in_X
18   affine_map<(d0,d1)->(d0)>],  // for argument %out_Y
19   iterator_types = ["parallel", "reduction"] }
20 %0  = linalg.generic #spmv ins(%A2, %in_X : tensor<?x?xf32, #BDIA_CSR>, tensor<?xf32>)
21     outs(%out_Y: tensor<?xf32>) {
22     ^bb0(%A_val: f32, %X_val: f32, %Y_val: f32):
23       %1  = arith.mulf %A_val, %X_val : f32
24       %o = arith.addf %Y_val, %1  : f32
25       linalg.yield %o : f32
26   } -> tensor<?xf32>
```

(d) A UniSparse program of the SpMV kernel with the format of the input tensor converted from COO to hybrid BDIA/CSR. The blocking size of the BDIA format is 3. File input operations are omitted. The **`decompose`** operation in Line 12 divides the tensor into two sub-tensors adaptively by embedding a **`sum`** primitive that queries the sparsity pattern of the tensor data. A **`convert`** operation in Line 13 translates the source into the target format. The SpMV kernel is specified using a **`linalg.generic`** operation in Line 21 - 26.

Fig. 3. An illustration of UniSparse.

(2) How to express the source and target format in a general way? Here we use COO and BDIA/CSR for illustration purposes only, while in practice, other formats can be used. (3) For productivity, conversion between the formats should be automated. How could our compiler figure out the memory layouts of the formats, and automatically convert one to the other? (4) For generality, it is desirable to write a compute kernel only once but the kernel works with a matrix in any format after conversion. How to decouple the compute kernel from a specific sparse format?

In Figure 3d, Line 2-9 specify the source and target formats. The sparsity pattern of the input is queried (Line 11), and the result is used to decompose the source format into two parts, both in the original source format COO (Line 12). Then, the decomposed source format is converted to the target format (Line 13). This target format is a parameter to the compute kernel (Line 15-26). In other words, the compute kernel and the format of its input matrix are completely decoupled. According to the above UniSparse program, the UniSparse compiler automatically infers the memory layout

of the source format (Figure 1b), and emits a sequence of internal operators (Figure 3c), which, once executed, result in the memory layout of the target format (Figure 1n).

4 TENSOR FORMAT ABSTRACTION

This section describes the proposed format abstraction that supports custom data structures and layouts for sparse tensors. We first describe the expression of sparse data structures in Section 4.1, which is logically represented as a metadata tree with index maps, query, and mutation primitives. The metadata tree representation and index maps are first introduced in TACO [Kjolstad et al. 2017; Chou et al. 2020a]. This work extends index maps to express a diverse range of custom formats. We further propose a new encoding method that holistically expresses data structures in primitives, without presuming per-dimension data structure separation. Section 4.2 shows how memory layouts are determined by applying layout primitives to the metadata tree, allowing formats to be expressed and customized at the physical layout level, which is also our new contribution.

4.1 Metadata Tree

To retain all the information from the original tensor, sparse formats store the metadata of non-zero elements explicitly. The structure of metadata determines the dimension order and intra-dimension coordinate arrangement, which dictate the structure of the corresponding value elements. The metadata structure is represented by a metadata tree that stacks tensor dimensions with the major one at the top. Value elements are attached at the bottom of the metadata tree, each associated with a path of the tree. In the following, we will introduce how the metadata tree is expressed by index maps (§4.1.1), query (§4.1.2), and mutation (§4.1.3) primitives. Figure 4 depicts the metadata trees of selected formats presented in Figure 1.

4.1.1 Index Map. We define the index map, denoted by $\mathcal{M}$, as a mapping from a tuple of logical dimension iterators to a tuple of destination index expressions that describe physical dimension iterators:

$$\mathcal{M} := (d_0, d_1, ..., d_n) \mapsto (e_0, e_1, ..., e_m)$$

where the logical indices $(d_0, d_1, ..., d_n)$ identify the dimensions of the original tensor, and the physical index expressions $(e_0, e_1, ..., e_m)$ serve as new dimension identifiers of the sparse format. The index map of a lossless sparse format has its reverse map $\mathcal{M}^{-1} := (e_0, e_1, ..., e_m) \mapsto (d_0, d_1, ..., d_n)$, which retrieves the dimension iterators of the original tensor.

Index maps affect the data structures of a sparse format by determining the major order of dimensions and the values of metadata. A simple example of an index map is `(d0, d1) -> (d1, d0)`, which represents a column-major matrix layout. In Figure 4, physical index expressions are associated with each level of the metadata tree for different formats in Figure 4a - 4i, where the original matrix in Figure 1a is indexed by `d0` and `d1`. For example, the CSB format in Figure 4g uses an index map of `(d0, d1) -> (d0/2, d1/2, d0%2, d1%2)`, where block ids `(d0/2, d1/2)` are at the major two levels, and `(d0%2, d1%2)` are inner block dimensions. Another example is the DIA format. The traditional DIA format (Figure 4e) uses an index map of `(d0, d1) -> (d1-d0, d0)`, which stores elements along the same diagonal with diagonal offsets expressed by `(d1-d0)` at the major dimension. By changing the second dimension of the `dst-index-list` in the traditional DIA format from `d0` to `d1`, a variant of the DIA format is obtained, as shown in Figure 4d. In the given example, the DIA-variant format is more space-efficient with `d1` ranging from 0 to 3 than the traditional DIA format with `d0` ranging from 0 to 4. Compared to TACO [Chou et al. 2018], which only supports the traditional DIA format by hardcoding level functions with types “range” and “offset”, our abstraction is more flexible and expressive by covering a wider range of formats such as DIA-variant via the index map.

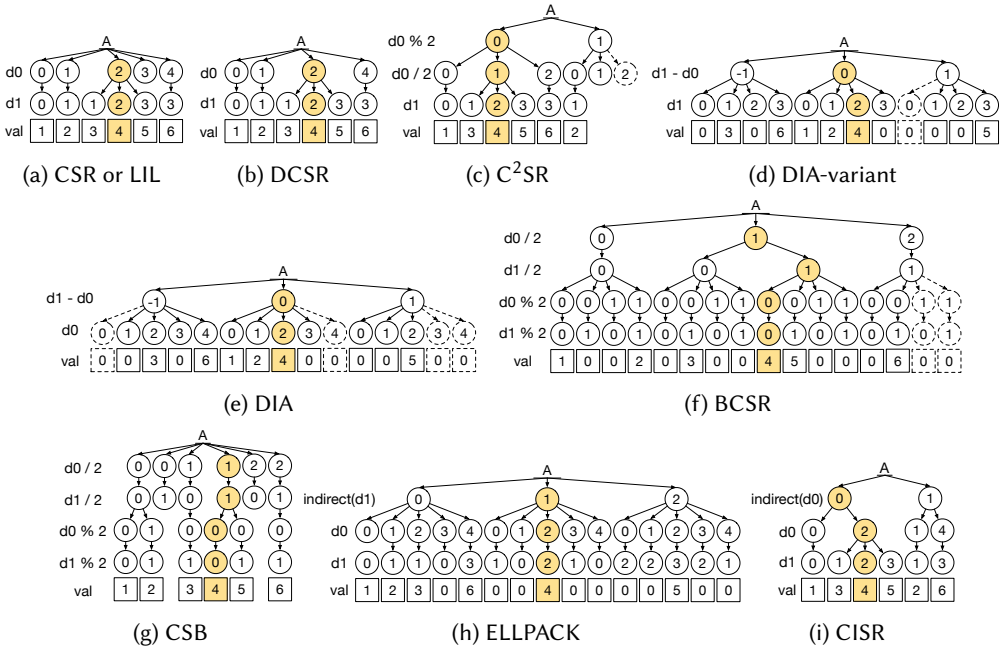


Fig. 4. The metadata trees of the matrix A in Figure 1a — Yellow-shaded blocks refer to the data structure of the same tensor element. The root of a metadata tree is the symbol of the tensor. A_0 and A_1 are sub-matrices of the matrix A. The circular nodes are metadata nodes that contain indices, and the arrows represent pointers from a higher major dimension to a lower one. Index expressions are on the left of each coordinate level. Nodes with dashed lines are elements padded for memory alignment.

```
1  index-map := src-index-list `->` dst-index-list
2  src-index-list := `(` id {`,` id} `)`
3  dst-index-list := `(` dst-expr {`,` dst-expr} `)`
4  dst-expr := id_expr
5              | `-`? integer-literal
6              | `-`? integer-literal `*` dst-expr
7              | dst-expr `+` dst-expr
8              | dst-expr `-` dst-expr
9              | id_expr `%` integer-literal
10             | id_expr `/` integer-literal
11 id_expr := id
12           | `indirect` dst-index-list
```

(a) The syntax of index maps.

```
1  query-primitive := query-func `(` var {`,` var}
       `)` {group-by-clause} {traverse-by-clause}
       {`with` val-map}
2  query-func := `sum` | `enum` | `reorder` |
       `schedule`
3  group-by-clause := `groupBy` index-map
4  traverse-by-clause := `traverseBy` index-map
5  val-map := {cond-val `->` integer-literal `|`}
       `otherwise` `->` integer-literal
6  cond-val := var `ne` | `eq` | `bt` | `be` | `lt`
       | `le` integer-literal
7  var := id | value
```

(b) The syntax for query primitives.

Fig. 5. The syntax for index maps and query primitives. `id` denotes dimension identifiers, `value` denotes tensor element values, and `integer-literal` denotes constant numbers. Expressions enclosed in curly braces can be repeated zero or more times.

The UniSparse syntax of index maps is shown in Figure 5a. The map takes a list of logical dimension indices as inputs and returns a list of arithmetic expressions that represent physical dimension iterators. The physical dimension iterators are typically expressed as closed-form functions using basic arithmetic operations (e.g., +, -, *, /, %) as shown in Line 4-10 of Figure 5a. We refer to index maps with pure arithmetic operations as *direct maps*. For generality, we further allow user-defined

custom functions, namely *indirect maps*, for the index iterator, rather than being limited to closed-form expressions. These dimension iterators are marked with the keyword `indirect`, followed by their parameter index terms (expressed as `dst-index-list`). Examples of *indirect maps* can be seen at the major dimensions of several formats, including ELLPACK (Figure 1k), CISR (Figure 1m), and the BELL portion of the hybrid BELL/COO (Figure 1o) format. In the next section, we introduce how indirect map functions are constructed in more detail.

4.1.2 Query Primitives. UniSparse provides methods to obtain statistics of a sparse tensor through query primitives. Each query primitive consists of a query function, a `group-by` clause, a `traverse-by` clause, and a `value map`. The syntax of query primitives is shown in Figure 5b.

The `group-by` clause uses an index map with usually fewer dimensions in the destination index expressions to divide tensor elements into groups. Commonly used operators include divide and modulo. For instance, the `group-by` function with map `(d0, d1) -> (d0%2)` assigns values in even rows to one group and values in odd rows to another group. Another map `(d0, d1) -> (d1-d0)` groups elements on the same diagonals together. The `traverse-by` clause specifies the traversing order of the indices with also an index map. For example, `traverse-by (d0, d1) -> (d0)` and `(d0, d1) -> (d1)` indicate traversing elements in increasing order of the row and column dimension identifier, respectively. The `value map` assigns a new value for each set of values satisfying a certain condition. For example, `value ne 0 -> 1 | otherwise -> 0` assigns non-zero elements to 1 and others to 0.

We predefine several query functions in Line 2: **`sum`**, **`enumerate`**, **`reorder`**, and **`schedule`**. The **`sum`** function computes the accumulation of values returned by the `value map` with a set of groups defined by the `group-by` clause. The **`enumerate`** assigns counting numbers with the start number defined by the `value map` in groups that follow the specified traversing order by `traverse-by`. The **`reorder`** returns the indices of the sorted elements with a specific traversing order. The **`schedule`** assigns tensor elements into a specified number of partitions in a balanced manner.

There are primarily two scenarios in UniSparse where query primitives are used. The first scenario involves supporting hybrid formats, where the **`sum`** function is used to query the non-zero distribution patterns before decomposing the input tensor. The other scenario involves supporting custom index maps with indirect mapping functions. Figure 7 illustrates how query primitives are used to construct indirect functions. Specifically, **`sum`** and **`enumerate`** are used to construct the indirect level of the ELL format in Lines 10. These functions assign counting numbers to non-zero and zero elements in increasing order. The **`sum`** and **`schedule`** functions constitute the indirect level in the CISR format (Line 14), which assigns rows to two buckets, each with a balanced number of non-zero elements.

4.1.3 Mutation Primitives. While index maps can be used to define new index values and change the dimension order, they do not take advantage of the sparsity to compress the data structures. In this section, we present primitives that enable the compression of tensor data structures. Specifically, we propose two mutation primitives – **`trim`**, which indicates the removal of tensor components associated with zero-values, and **`merge`**, which expresses the reduction of replicated components. We also introduce conversion operators related to the primitives **`trim`** and **`merge`** to better illustrate how the expression of one format can be changed to another format by making slight modifications to its encoding. Note that these conversion operators are distinct from encoding primitives used to describe formats. To differentiate between them, we name conversion operators in CamelCase notation.

Trim. The **`trim`** primitive takes two numerical values representing dimension levels, where the number 0 corresponds to the major dimension at the top of the metadata hierarchy, and subsequent levels increase from top to bottom. The primitive **`trim`**`(S,E)` indicates the removal of zero values (at

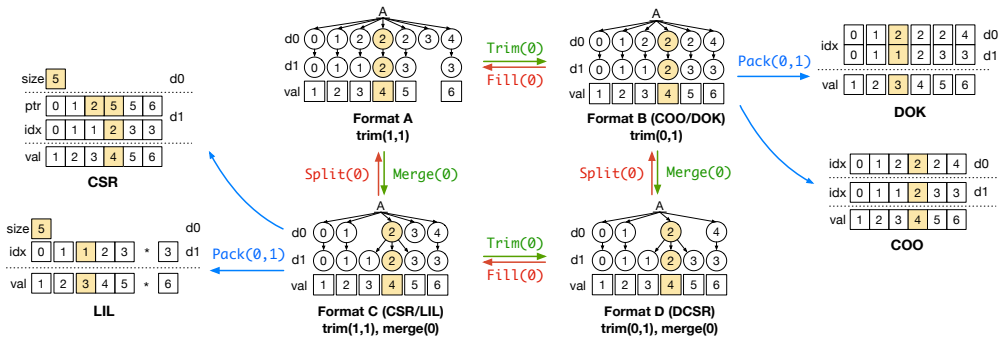


Fig. 6. An illustration of mutation primitives and layout primitives. The formats A, B, C, and D of the matrix A in Figure 1a can be described using different combinations of the **merge** and **trim** primitives. Each format can be transformed into its neighbor via a conversion operator indicated by the arrows. Green arrows denote sparsification directions, and red arrows denote densification directions. Notice that the two conversion operators `Trim` and `Merge` are orthogonal and commutative. Blue arrows point to different physical layouts of the formats.

the bottom of the tree) and the associated metadata (on the path from the root to leaves) between a starting level `S` and an ending level `E`, where `S`≤`E` (i.e., `S` is closer to the root than `E`). Specifically, **`trim`**`(S,E)` first removes data nodes belonging to the sub-tensor identified by `(d0, d1, ..., dE)`, if all elements in that sub-tensor are zeros. Then it cleans up dangling nodes that have no children nodes from level `E` up to level `S`. For example, in formats B and D of Figure 6, the primitive **`trim`**`(0,1)` indicates that all zero values identified by dimensions `(d0, d1)` are removed, along with all their metadata. In contrast, the primitive **`trim`**`(1,1)` in formats A and C only removes zero values and their column indices at the least major dimension `(d1)`, while leaving dangling nodes (row index 3) at dimension `(d0)`.

The conversion operator `Trim(L)` implements the removal of metadata at level `L` if the sub-tensor identified by indices at level `L` contains only zeros. The reverse operator `Fill(L)` inserts nodes at level `L` for all existing parents by creating nodes with missing index values in the range of `[0,DL)` where `DL` is the dimension size at level `L`. Figure 6 shows how `Fill(0)` converts format B to A and D to C by adding index nodes for the empty row.

Merge. The **`merge`** primitive reduces storage space by merging equivalent paths at specified levels. It takes in a list of numerical values representing dimension levels, and for each level `L` that has **`merge`** applied, all repeating paths from the top-level dimension down to level `L` will be fused into one path. Children nodes at level `L+1` sharing the same metadata are inherited by the fused parent node. In Figure 6, formats C and D have the encoding **`merge`**`(0)`, which indicates fusing repetitive nodes at dimension `(d0)`.

The conversion operator `Merge(L)` implements the fusion of repetitive metadata nodes at level `L`. The reverse operator `Split(L)` restores multiple copies of parent paths from the root to level `L` for nodes with more than one child at level `L`. In Figure 6, applying `Split(0)` converts format C to A and D to B by replicating nodes at the major dimension, ensuring that each row index node has only one child node.

4.2 Data Layout

A metadata tree can be realized using different physical memory layouts. By default, it is stored in an SoA layout, where each dimension is stored using separate arrays. UniSparse further provides two primitives to express different memory layout options. The **`partition`** primitive determines

```
#COO = #encoding<{ idx_map<(d0,d1)->(d0,d1)>, mutation<trim(0,1)> }>
#DOK = #encoding<{ idx_map<(d0,d1)->(d0,d1)>, mutation<trim(0,1)>, layout<pack(0,1)> }>
#CSR = #encoding<{ idx_map<(d0,d1)->(d0,d1)>, mutation<merge(0), trim(1,1)> }>
#DCSR = #encoding<{ idx_map<(d0,d1)->(d0,d1)>, mutation<merge(0), trim(0,1)> }>
#DIA = #encoding<{ idx_map<(d0,d1)->(d1-d0,d0)>, mutation<merge(0), trim(0,0)> }>
#DIA-variant = #encoding<{ idx_map<(d0,d1)->(d1-d0,d1)>, mutation<merge(0), trim(0,0)> }>
#BCSR = #encoding<{ idx_map<(d0,d1)->(d0/2,d1/2,d0%2,d1%2)>, mutation<merge(0,1), trim(1,1)> }>
#CSB = #encoding<{ idx_map<(d0,d1)->(d0/2,d1/2,d0%2,d1%2)>, mutation<merge(1), trim(2,3)> }>
#ELL = #encoding<{ idx_map<(d0,d1)->(indirect(d1),d0,d1)>, mutation<merge(0), trim(0,0)>,
  indirect<{ sum(value) groupBy (d0,d1)->(d0) with value ne 0 -> 1 | otherwise -> 0
             enum(value) groupBy (d0,d1)->(d0) traverseBy (d0,d1)->(d1) with value eq 0 -> sumVal |
      otherwise -> 0 }> }>
#C2SR = #encoding<{ idx_map<(d0,d1)->(d0%2,d0/2,d1)>, mutation<merge(0,1), trim(2,2)>, layout<partition(0)>}>
#CISR = #encoding<{ idx_map<(d0,d1)->(indirect(d0),d0,d1)>, mutation<merge(0,1), trim(1,2)>,
  indirect<{ sum(value) groupBy (d0,d1)->(d0) with value ne 0 -> 1 | otherwise -> 0
             schedule(d0) traverseBy (d0,d1)->(d0/2) }>,
  layout<partition(0)> }>
```

Fig. 7. Encodings of selected sparse formats in Figure 1 written in pseudo MLIR code.

whether to divide one tensor into several sub-tensors, which can be beneficial for storing the sparse tensor in banked memories. The **pack** primitive switches from the SoA layout to an AoS layout for a single tensor, providing a larger design space with more layout options for users to explore.

Partition. The **partition** primitive splits metadata with their descendant nodes at the specified dimension level into subtrees, each stored separately as a sub-tensor. This is particularly useful for multi-bank memory systems, where each sub-tensor can be stored in a different bank to reduce memory access conflicts and improve performance. For example, the C^2SR and CISR formats are partitioned into sub-tensors with **partition**(0), and each sub-tensor is designed to occupy a memory bank. The format encodings of the C^2SR and CISR formats are shown in Figure 7.

Pack. The **pack** primitive enables the creation of an AoS layout for a tensor by specifying two dimension levels, S and E, and enforcing a depth-first traversal of the metadata tree for dimensions from S to E. By default, a breadth-first traversal of the metadata tree generates level-wise arrays which correspond to an SoA format. For example, in Figure 6, a breadth-first traversal of the metadata tree B results in {d0=[0,1,2,2,2,4], d1=[0,1,1,2,3,3], val=[1,2,3,4,5,6]}, which corresponds to the COO format. Similarly, the default traversal order of the metadata tree C leads to an SoA layout, which is the CSR format. On the other hand, adding the primitive **pack**(0,1) to the metadata tree B pairs up index values at dimensions (d0, d1) and generates an AoS layout, i.e., [{d0=0, d1=0, val=1}, {d0=1, d1=1, val=2}, ..., {d0=4, d1=3, val=6}], corresponding to the DOK format. Similarly, adding the primitive **pack**(0,1) to format C generates the AoS layout counterpart of the CSR format, which is the LIL format.

With the above primitives, UniSparse can express a wide range of custom formats, including those with custom index maps (e.g., load-balanced formats in Figure 1m), sparsity pattern-aware hybrid formats (e.g., Figure 1n/1o), and backend-aware memory layouts (e.g., banked formats in Figure 1l/1m). Figure 7 lists the abstractions of all the formats in Figure 1.

5 COMPILATION

To generate code for format customization and compute operations, the UniSparse compiler first decodes the data structures and layouts from format descriptions specified in the intermediate language (§5.1). Once the source and destination formats are determined, the compiler lowers the **convert** operation by applying a sequence of rewrite rules and emitting conversion operators that gradually convert from the source format to the destination format (§5.2). For compute operation lowering, UniSparse leverages the MLIR SparseTensor dialect to handle classic formats with the

index map `(d0, d1)->(d0, d1)`. In the case of custom formats, UniSparse implements a general compute kernel generation algorithm (§5.3).

5.1 Inference of Data Structure and Layout

The UniSparse compiler infers the data structures and layouts, like those shown in Figure 1, from the formats specified in the UniSparse language (Figure 7). The conservative approach for storing a sparse format involves maintaining an array of indices (**`idx`**) and an array of pointers (**`ptr`**) at each level of the metadata tree. An element in an **`idx`** array identifies a node at the current tree level. An element in a **`ptr`** array indicates how many nodes are connected to a parent node, i.e., it encodes a down arrow ↓ in Figure 4. However, the storage can be simplified in special cases. For example, when the indices at the current level are contiguous numbers starting from 0, the **`idx`** array can simply be replaced with a **`size`**. In another case where nodes have a one-to-one correspondence with their parents, the **`ptr`** array can be skipped.

Based on these principles, the compiler infers the simplified data structures of a sparse format. A dimension level with no **`trim`** or **`merge`** applied is stored in **`size`**. The **`trim`**`(S,E)` primitive requires explicit **`idx`** arrays at dimensions from `S` to level `E`, as indices at these levels are no longer contiguous after being trimmed. The **`merge`** primitive adds a **`ptr`** array to the descendant levels of the specified levels, if the descendant levels also have **`trim`** applied, since the number of children nodes varies from one parent node to another at the level being merged. For example, to store the CSR format, an **`idx`** array is required at the column dimension because it is trimmed, and a **`ptr`** array is also needed at the column dimension, as the parent dimension (i.e., the row dimension) is merged. The DCSR format requires an **`idx`** array at both the row and column dimensions since they are both trimmed. The DIA format requires an **`idx`** array at the major dimension, which stores the diagonal offsets. Although the major dimension has been merged, it does not add a **`ptr`** array to its descendant level since the second dimension is not trimmed. Indirect maps introduce less regular index patterns, which also require an explicit **`idx`** array.

The aforementioned steps infer the data structures of a sparse format, but the actual physical layout is not yet determined. In UniSparse, the physical memory layout of a sparse tensor can be customized using the **`partition`** and **`pack`** primitives. The **`partition`** primitive divides indices and all their descendant metadata at the specified level into sub-tensors. For example, the C^2SR format stores sub-tensors separately in memory banks, with the **`partition`** primitive applied at the major level. On the other hand, the **`pack`**`(S,E)` primitive generates an AoS layout of the tensor from level `S` to level `E`, as opposed to the default SoA layout. The value array at the bottom of the metadata tree, which is treated as an additional level of data, can also be packed with metadata. If the packed levels have the same number of elements, they are stored in an array of tuples, such as the DOK format. Alternatively, the elements can be stored in an array of variable-length lists, such as the LIL format.

5.2 Format Conversion

The UniSparse compiler incorporates a general algorithm that automates conversion between any two custom formats specified with mutation primitives and index maps that contain purely arithmetic operations `{+, -, *, %, /}`. Indirect functions and layout primitives are not reversible, and therefore, UniSparse does not support automated conversion when the *source* format contains indirect functions or layout primitives. However, they are allowed in the target format.

The format conversion algorithm within UniSparse involves a thorough analysis of both the source and destination format encoding. It proceeds by individually conducting pattern matching on each component of the source and target format encoding. This process applies a series of

rewrite rules to convert the source format step-by-step to the target format. These rewrite rules are implemented as conversion operators that transform data structures and layouts in atomic steps.

Table 2 presents a collection of rewrite rules along with their corresponding conversion operators. These rewrite rules can be categorized into four sets: the rules for rewriting index maps including affine (①, ②, and ③) and non-affine (④ and ⑤) transformations, for matching mutation primitives (⑥-⑪), for querying (⑫-⑮) and for layout transformation (⑯-⑰).

Affine transformations are equivalently represented as matrices in the table. For instance, consider rule ①, which matches the case when two dimensions in the source format switch their positions in the target format. The effect of rule ① is expressed through an elementary transformation matrix as shown in the "Matrix" column of the table, and the operator `Swap` updates the data structures in the target metadata tree. Non-affine transformation rules are applicable when there are `` `/` `` and `` `%` `` in the source or target index map. These non-affine transformations are implemented using the `TileUnion` and `TileSplit` operators.

Rule ⑥-⑪ handle the situations when either the source or the target format contains mutation primitives. For example, if dimension S to dimension E are trimmed in the source format, but the preceding dimension $S-1$ is additionally trimmed in the target format, the corresponding conversion operator should perform this additional trim (Rule ⑥). Conversely, if the target format has one less dimension trimmed than the source format, the corresponding conversion operator should fill that dimension (Rule ⑦). Similarly, the formats could differ in the last trimmed dimension, and these cases are handled by Rule ⑧ and ⑨. These two rules also cover the special situation when $S == E$, in which case one format contains **`trim`** and the other does not. Furthermore, Rule ⑩ merges one more dimension based on the source format, and Rule ⑪ removes one merged dimension.

Finally, Rule ⑫-⑰ directly work on the target format when the target format contains query or layout primitives. Note that these two sets of rewrite rules, as well as the encoding primitives, are open-ended and can be extended when necessary to support new user-defined custom formats.

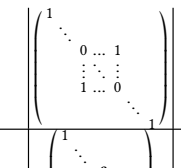
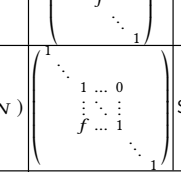
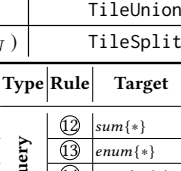
The rewrite rules presented in Table 2 ensure that UniSparse is fully capable of converting formats encoded only with direct index maps and mutation primitives (referred to as set S) into arbitrary formats expressed in UniSparse notation (referred to as set G). To demonstrate the completeness of format conversion in UniSparse, we first establish its ability to convert between any two formats within set S using the rewrite rules provided in Table 2. Then, we discuss the conversion from a format in set S to a format in set $G-S$ that involves indirect functions and layout primitives.

We discuss the conversion between arbitrary formats in set S in two steps.

(i) *Reversibility*. The conversion process between formats within set S is reversible, implying that for any two formats, A and B, encoded solely using direct index maps and mutation primitives, if there exists a sequence of conversion operators transforming format A to format B, there must also exist a conversion path that can transform format B back to format A. This reversibility property is guaranteed by the conversion operators outlined in Table 2. Specifically, Table 2 contains arithmetic conversion operators, each having its reverse pair: `Swap(i, j)` and `Swap(j, i)`, `Scale(i, f)` and `Scale(i, 1/f)`, `Skew(i, j, f)` and `Skew(i, j, -f)`, `TileUnion(i, f)` and `TileSplit(i, f)`. Furthermore, the conversion operators for rule ⑥ and ⑦, those for rule ⑧ and ⑨, and rule ⑩ and rule ⑪ also form reverse pairs.

(ii) *Reachability*. We further show that there exists a reference format, denoted as R, from which all formats within set S can be reached, meaning they can be converted from R. Without loss of generality, we select the COO format as R. Given the reversibility property of format conversion within set S, we only need to show that any arbitrary format within set S can be converted into

Table 2. Rewrite rules and conversion operators that support the UniSparse format conversion algorithm. Parameters of query primitives are omitted as {*} due to the limited length of the table. The syntax of the query primitives is shown in Figure 5b, and examples can be found in Figure 7.

Type	Rule	Source	Target	Conversion: Matrix	Conversion: Operator
Index Map	①	$(d_0, ..., d_i, ..., d_j, ..., d_N)$	$(d_0, ..., d_j, ..., d_i, ..., d_N)$		`Swap(i,j)`
	②	$(d_0, ..., d_i, ..., d_N)$	$(d_0, ..., f*d_i, ..., d_N)$		`Scale(i,f)`
	③	$(d_0, ..., d_i, ..., d_j, ..., d_N)$	$(d_0, ..., d_i, ..., f*d_i+d_j, ..., d_N)$		`Skew(i,j,f)`
	④	$(d_0, ..., d_i / f, d_i \% f, ..., d_N)$	$(d_0, ..., d_i, ..., d_N)$	`TileUnion(i,f)`	
	⑤	$(d_0, ..., d_i, ..., d_N)$	$(d_0, ..., d_i / f, d_i \% f, ..., d_N)$	`TileSplit(i,f)`	

Type	Rule	Source	Target	Conversion	Type	Rule	Target	Conversion
Mutation	⑥	$trim(d_S, d_E), S \le E$	$trim(d_{S-1}, d_E), S \le E$	`Trim(S-1)`	Query	⑫	$sum\{*\}$	`Sum()`
	⑦	$trim(d_{S-1}, d_E), S \le E$	$trim(d_S, d_E), S \le E$	`Fill(S-1)`		⑬	$enum\{*\}$	`Enumerate()`
	⑧	$trim(d_S, d_{E-1}), S \le E$	$trim(d_S, d_E), S \le E$	`Devectorize(E), Trim(E), Vectorize(E+1)`		⑭	$reorder\{*\}$	`Reorder()`
						⑮	$schedule\{*\}$	`Schedule()`
	⑨	$trim(d_S, d_E), S \le E$	$trim(d_S, d_{E-1}), S \le E$	`Devectorize(E+1), Fill(E), Vectorize(E)`	Layout	⑯	$pack(d_S, d_E)$	`Pack(S, E)`
	⑩	$merge(d_0, ..., d_{i-1})$	$merge(d_0, ..., d_{i-1}, d_i)$	`Merge(i)`		⑰	$partition(d_i)$	`Partition(i)`
	⑪	$merge(d_0, ..., d_{i-1}, d_i)$	$merge(d_0, ..., d_{i-1})$	`Split(i)`				

the COO format. This can be achieved through the following observations: Rule ⑤ eliminates pairs of divide and modulo operations, ensuring that any formats within set S can be converted into those without divide and modulo operations in the index maps; rules ① through ③ cover the entire affine transformation space, allowing for the conversion of affine index expressions into the original dimension identifiers in the COO format; and rule ⑪ removes merged levels, while rules ⑥ and ⑧ expand trimmed levels, enabling the conversion of any mutation primitives into the encoding form of COO format.

Combining the two key points (i) and (ii), we can conclude that UniSparse can convert between any format within set S using the rewrite rules and conversion operators in Table 2.

For the conversion from a format within set S to a format within set $G - S$, which includes indirect functions and layout primitives, one can directly apply rules ⑫ through ⑰ that align a source format with the target format's encoding. Building upon the earlier discussion on format conversion within set S, we know that if there exists one format within set S capable of converting to any format within $G - S$, then any arbitrary format from set S will have the ability to convert to any format in $G - S$ as well. As a result, UniSparse can perform conversions from any format in set S to any format in set G.

Figure 8 shows the pseudocode of our format conversion algorithm, which includes three steps. Initially, the compiler aligns the source index map with the target index map (Step 1), followed by the mutation of data structures (Step 2). Finally, the memory layout is generated (Step 3).

```
1  Algorithm: Format Conversion
2  Input: source format encoding S, target format encoding T, source format storage S_format
3  Output: target format storage T_format
4  FormatConversion {
5    T_format = S_format // directly manipulate the source format storage
6    if index_map of S != index_map of T: // step 1: Index Map Alignment
7      if index_map of S contains (di/f, di%f):
8        apply Rule ④ to T_format;
9      transMatrix = indexMapMatrix(S)^-1 * indexMapMatrix(T);
10     for each non-identity sub-matrix E of transMatrix:
11       apply E to T_format; // Rule ①,②,③ in Table 2
12     if index_map of T contains (di/f, di%f):
13       apply Rule ⑤ to T_format;
14     for each indirect_function f of {sum, enum, reorder, schedule} in T:
15       apply f to T_format; // Rule ⑫-⑮ in Table 2
16     apply Sort to T_format; // sort indices
17   while trim of S != trim of T: // step 2: Structure Mutation
18     apply Rule ⑥/⑦/⑧/⑨ to T_format;
19   while merge of S != merge of T:
20     apply Rule ⑩/⑪ to T_format;
21   for each layout l of {pack, partition} in T: // step 3: Layout Generation
22     apply l to T_format;  // Rule ⑯,⑰ in Table 2
23   return T_format;
24 }
```

Fig. 8. The pseudo-code of the UniSparse format conversion algorithm.

In the first step, new indices are calculated for the target format (line 6 - 16). If the source format encoding **S** contains `/` and `%` operations, the compiler employs `TileUnion` to eliminate these non-affine operations and generate an affine index map of **S** (line 7 - 8). Next, the compiler computes the transformation matrix `transMatrix` by multiplying the inverse matrix of the source affine index map with the matrix of the target affine index map (line 9). The function `indexMapMatrix` outputs the matrix representation of an affine index map, e.g., `(1, 1; -1, 1)` for `(d0+d1, d1-d0)`. By breaking down the transformation matrix into elementary transformation matrices (line 10), the corresponding conversion operators are sequentially applied to the source format (line 11). Following this, the compiler applies `TileSplit` if the target index map contains tiling (line 12 - 13). The handling of indirect functions (line 14 - 15) adheres to the encoding sequence of the target format. Now all the target indices are ready, and a `Sort` step orders the indices as specified in the target format. In the second step, the compiler addresses the disparities between **`trim`** and **`merge`** primitives in the source and target formats, thereby mutating data structures (line 17 - 20) using the conversion rules ⑥ - ⑪ outlined in Table 2. In the final step, the compiler handles the **`pack`** and **`partition`** in the target format and generates custom layouts (line 21 - 22).

Examples of conversions from COO to BDIA and from COO to CSR are shown in Figure 3c. When converting from COO to BDIA format, the process begins with the application of index map rules to calculate a new set of index iterators in the target format: `Skew(0, 1, -1)` transforms the COO index map `(d0, d1)` into `(d0, d1-d0)`, and `TileSplit(0, 3)` subsequently divides `d0` into tiling dimensions `d0/3` and `d0%3`. To ensure the proper metadata order `(d0/3, d1-d0, d0%3)`, the `Sort` operator is employed. Following index map alignment, the two-dimensional format COO turns into a three-dimensional intermediate format with a mutation primitive **`trim`**`(0,1)`. To convert the intermediate format into the target BDIA format that would contain {**`merge`**`(0)`, **`trim`**`(1,1)`}, `Fill(0)` is applied to obtain **`trim`**`(1,1)`, and `Merge(0)` is used to derive **`merge`**`(0)`. In the case of COO to CSR format conversion, the distinction lies only in their mutation primitives. Therefore, converting from COO to CSR involves just two operators: `Fill(0)`, which matches the pattern converting from **`trim`**`(0,1)` in COO to **`trim`**`(1,1)` in CSR, and `Merge(0)`, which transforms the absence of merge in COO to **`merge`**`(0)` in CSR.

5.3 Compute Kernel Generation

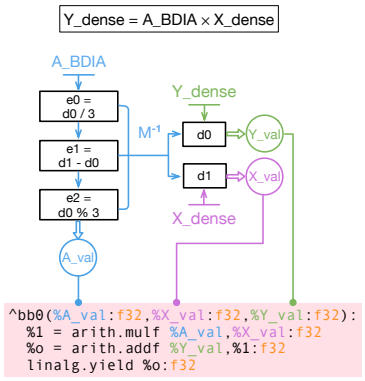
UniSparse extends the MLIR SparseTensor dialect, which leverages TACO's code generation algorithms, to generate compute kernels for conventional formats with the index map `(d0, d1) -> (d0, d1)`. When dealing with formats with custom index maps that are unsupported by MLIR SparseTensor, UniSparse adopts a two-step code generation algorithm, where the compiler generates a functionally correct code in the first step, and optimizes the generated code for better performance in the second step. Figure 9 illustrates the UniSparse kernel generation algorithm using the SpMV kernel in BDIA format as an example.

In the first step, the UniSparse compiler generates a compute kernel that conducts exhaustive iterations across all dimensions of all operands (i.e., tensors of both inputs and output) and performs computations on values with matching indices. The rationale behind generating an exhaustive iteration across all metadata dimensions is rooted in the fact that different physical dimension iterators of tensor operands can not be co-iterated. For instance, consider the SpMV case with the BDIA format, as illustrated in Figure 9a. Here, the matrix A features dimension iterators expressed as `(e0=d0/3, e1=d1-d0, e2=d0%3)`, while the input vector has iterator `(d0)` and the output vector has iterator `(d1)`. There exist five distinct physical dimension iterators `{e0, e1, e2, d0, d1}`, but no pairs can be iterated together. Therefore, as illustrated in Figure 9b, the UniSparse compiler generates a loop nest that iterates through all the dimensions of the input tensor (line 1 - 4), the input vector (line 5), as well as the output vector (line 6). The iteration through the dimensions of the tensor corresponds to traversing the metadata tree. Progressing from one tree level (i.e., a tensor dimension) to the next level requires traversing a single tree edge, which may involve memory accesses. In this example, there are 3 tree levels (see the BDIA example in Figure 1n): the first and last levels are encoded by two sizes, each visited by a single loop (lines 1 and 4, respectively), and the indices of the tensor for these two levels are simply the loop variables (line 7 and 9); however, the middle level is encoded by an index array and a pointer array, requiring it to be visited in two loops (line 2-3), and the tensor's index for this level is retrieved from the pointer array via a memory access (line 8).

Now that the indices of the tensor in BDIA format have been retrieved, the logical indices can be restored (lines 10 - 11). In general, the UniSparse compiler employs the reverse transformation $\mathcal{M}^{-1}$ to restore the logical indices, as illustrated in Figure 9a. In this specific example, $\mathcal{M}^{-1}$ comprises the transformations `d0=(d0/3)*3+(d0%3)` and `d1=(d1-d0)+d0`. Once the logical indices are recovered, the compiler generates code to verify whether these logical indices fall within valid boundaries (line 14), and ensures the index values of the contraction dimensions match (line 15). In this case, the contraction dimension is `d1`, so the generated code checks for the equality of index `d1` shared by the matrix A and the input vector X. Inside the loop body, tensor element values are accessed (lines 16 - 18), followed by computation (line 19) which is lowered from the program specification in Figure 9a, and the results are written into the output tensor (line 20).

In the second step, the UniSparse compiler removes redundant iterations from the generated kernel for better performance. To merge iterations, the UniSparse compiler checks for several cases:

- **a.** Co-iteration within a single tensor – consecutively trimmed dimensions contain metadata of the same length, enabling them to be co-iterated. For instance, in the COO format with **`trim`**`(0,1)`, both the row and column dimensions contain index arrays of identical lengths, allowing them to be iterated together using a single loop.
- **b.** Co-iteration across multiple tensors –
  Case I. The contraction dimension of multiple tensors can be co-iterated if they share the same physical dimension iterator. For example, when multiplying DIA (index map `(d0, d1) -> (d1-d0, d0)`) with DCSR (index map `(d0, d1) -> (d0, d1)`), the second dimension of DIA



(a) Logical index matching among tensor operands.

```
1  for id0 in range(0,bdia_size_0):
2    for id_1_0 in range(0,bdia_ptr_1.len()-1):
3      for id_1_1 in range(bdia_ptr_1[id_1_0],bdia_ptr_1[id_1_0+1]):
4        for id2 in range(0,bdia_size_2):
5          for x_id in range(0,X_dense.len()): // step 2: remove
6            for y_id in range(0,Y_dense.len()): // step 2: remove
7              bdia_e0 = id0
8              bdia_e1 = bdia_idx_1[id_1_1]
9              bdia_e2 = id2
10             bdia_d0 = bdia_e0*3 + bdia_e2
11             bdia_d1 = bdia_e1 + bdia_d0
12             X_dense_d1 = x_id // step 2: X_dense_d1 = bdia_d1
13             Y_dense_d0 = y_id // step 2: Y_dense_d0 = bdia_d0
14             if (bdia_d0 in range(0,D0) and bdia_d1 in range(0,D1)
15                 and bdia_d1 == X_dense_d1): // step 2: remove
16               A_val = bdia_val[id_1_1 * bdia_size_2 + id2]
17               X_val = X_dense[X_dense_d1]
18               Y_val = Y_dense[Y_dense_d0]
19               res = Y_val + A_val*X_val
20               Y_dense[Y_dense_d0] = res
```

(b) The pseudo-code of the generated SpMV kernel.

Fig. 9. Exemplifying the UniSparse kernel generation algorithm with the SpMV kernel using BDIA format.

and the row dimension of DCSR share the physical iterator `d0`, allowing these two dimensions to be co-iterated as `while (dia_id1 < dia_id1.len() and dcsr_id0 < dcsr_idx_0.len())`.

Case II. Dimensions that support random access can reuse the shared iterator of other tensors. For example, in Figure 9, the input and output vectors are dense arrays that can be randomly accessed, enabling them to directly borrow the logical index iterators of the matrix A. Consequently, the compiler eliminates iterations in lines 5-6 and the index equality check in line 15.

6 THE UNISPARSE MLIR DIALECT

The UniSparse intermediate language is implemented as a standalone MLIR dialect. Figure 3d shows a UniSparse program of the SpMV kernel with the hybrid BDIA/CSR format. This section introduces the key ingredients of the UniSparse language:

Types and Attributes. Format descriptions are specified as MLIR attributes. In the example program shown in Figure 3d, all formats are specified beforehand using index maps (idx_map) and mutation primitives (mutation) in Line 2-9. Sparse tensors are declared with `tensor` types and sparse format encoding attributes. Annotations specify the desired sparse format, eliminating the need for explicit handling of sparsity in code.

Tensor Preprocessing Operations. The UniSparse MLIR dialect defines two key operations for sparse format customization: **`decompose`** and **`convert`**. The **`decompose`** operation splits the input tensor into sub-tensors with different sparsity ranges, based on the non-zero distribution patterns (identified by the **`sum`** primitive).The **`convert`** operation specifies format conversion between source and destination formats based on their respective encodings.

Compute Kernels. The compute kernel is specified using the **`linalg.generic`** operation within MLIR. Figure 3d (lines 15-24) provides an example of the **`linalg.generic`** operation that specifies an SpMV kernel. Within this operation, the #spmv attribute (lines 15-19) indicates the logical dimension iterators and compute patterns of tensor operands. The inputs and outputs (lines 20-21) of the **`linalg.generic`** operation define the iteration space, while the loop body (lines 23-24) details the computational logic.

Table 3. A summary of the matrices used for evaluation with the abbreviated names in parentheses.

Matrix	Shape	Density	Nonzero Diagonals	Matrix	Shape	Density	Nonzero Diagonals
email-Eu-core (ee)	1.01K × 1.01K	2.5e-2	1.84K	ML_Geer (ge)	1.50M × 1.50M	4.9e-5	9.35K
ss (ss)	1.65M × 1.65M	1.3e-5	1.68M	ML_Laplace (lp)	377K × 377K	1.9e-4	4.70K
Transport (tp)	1.60M × 1.60M	9.2e-6	15	rajat31 (ra)	4.69M × 4.69M	9.2e-7	5.05K
TSOPF_RS_b2383 (ts)	38.1K × 38.1K	1.1e-2	23.4K	memchip (mc)	2.71M × 2.71M	2.0e-6	1.74M
vas_stokes_1M (vs)	1.09M × 1.09M	2.9e-5	1.67M	crystm02 (cm)	14.0K × 14.0K	1.7e-3	27
cant (ct)	62.5K × 62.5K	1.0e-3	99	c8_mat11 (c8)	4.56K × 5.76K	9.4e-2	10.3K
nemeth21 (nm)	9.51K × 9.51K	1.3e-2	169	heart1 (h1)	3.56K × 3.56K	1.1e-1	6.53K
bibd_18_9 (b9)	153 × 48.6K	2.4e-1	48.6K	cari (cr)	400 × 1.20K	3.2e-1	1.16K
transformer-0.5 (tf-0.5)	512 × 33.3K	5.0e-1	33.8K	transformer-0.6 (tf-0.6)	512 × 33.3K	4.0e-1	33.8K
transformer-0.7 (tf-0.7)	512 × 33.3K	3.0e-1	33.8K	transformer-0.8 (tf-0.8)	512 × 33.3K	2.0e-1	33.8K
transformer-0.9 (tf-0.9)	512 × 33.3K	1.0e-1	33.8K	transformer-0.95 (tf-0.95)	512 × 33.3K	5.0e-2	33.7K
roadNet-PA (rp)	1.09M × 1.09M	1.3e-6	66.4K	mouse_gene (mg)	45.1K × 45.1K	7.1e-3	77.8K
google-plus (gp)	108K × 108K	1.2e-3	203K	pokec (pk)	1.63M × 1.63M	1.2e-5	2.28M
hollywood (hw)	1.07M × 1.07M	4.9e-5	2.08M	ogbl-ppa (op)	576K × 576K	1.3e-4	1.14M
LiveJournal (lj)	4.85M × 4.85M	2.9e-6	6.55M	wikipedia-20051105 (wp)	1.63M × 1.63M	7.4e-6	2.86M
chem_master1 (ch)	40.4K × 40.4K	1.2e-5	5	majorbasis (mj)	160K × 160K	6.8e-5	22
shyy161 (sh)	76.5K × 76.5K	5.6e-5	7	Baumann (bm)	112K × 112K	5.9e-5	7
wiki-Vote (wv)	8.30K × 8.30K	1.5e-3	11.4K	mario002 (m2)	390K × 390K	1.4e-5	507K
scircuit (sc)	171K × 171K	3.3e-5	159K	p2pGnutella31 (pg)	62.6K × 62.6K	3.8e-5	53.2K
cage12 (ca)	130K × 130K	1.2e-4	75.5K	filter3D (f3)	106K × 106K	2.4e-4	13.4K
ca-CondMat (cc)	23.1K × 23.1K	3.5e-4	40.2K	poisson3Da (p3)	13.5K × 13.5K	1.9e-3	26.1K
bwm2000 (bw)	2.00K × 2.00K	2.0e-3	5	af23560 (af)	23.6K × 23.6K	8.7e-4	33
cryg10000 (cg)	10.0K × 10.0K	5.0e-4	8	ex19 (ex)	12.0K × 12.0K	1.8e-3	185
mycielskian12 (my)	3.07K × 3.07K	4.3e-2	6.12K	ogbl-ddi (od)	4.27K × 4.27K	5.8e-2	8.50K

7 EVALUATION

This section demonstrates the efficacy of UniSparse through a series of case studies (§7.2). These case studies illustrate how the adoption of custom formats enabled by UniSparse leads to improved performance for common sparse linear algebra kernels across a variety of hardware platforms, including an Intel multi-core CPU, an NVIDIA GPU, an AMD Xilinx FPGA, and a simulated PIM device. Furthermore, we assess the performance of automatic format conversion (§7.3) and compute kernel generation (§7.4) using UniSparse. Our evaluation demonstrates that the programs generated by UniSparse achieve performance that matches state-of-the-art approaches [Bik et al. 2022; Chou et al. 2020a], while UniSparse offers broader coverage for handling a wider range of custom formats.

7.1 Experiment Setup

We obtain sparse matrices from various popular datasets, including SuiteSparse [Davis and Hu 2011], SNAP [Leskovec and Krevl 2014], and OGB [Hu et al. 2020], covering a rich mix of application domains. Additionally, we also collect a set of sparse weight tensors/matrices from a pruned Transformer model [Gale et al. 2019]. Table 3 summarizes all the sparse matrices used in our experiments. For improved readability, we use abbreviated names for these matrices in all the illustrations presented below.

All format conversion experiments are performed on a dual-socket 24C/24T Intel(R) Xeon(R) Gold 6248R CPU @ 3.00 GHz. The compute kernels generated by UniSparse are executed on multiple backends, including the same CPU, an NVIDIA GPU A6000, an AMD Xilinx FPGA, and a PIM-core simulator [Devic et al. 2022]. We provide more details of the configurations for each experiment in the corresponding subsections.

7.2 Format Customization

We present four case studies evaluating the format customization feature of UniSparse. Our results demonstrate performance improvements on the CPU and the GPU using hybrid BDIA/CSR (§7.2.1) and hybrid BELL/COO (§7.2.2) formats that are tailored to the non-zero distribution patterns of

matrices. We showcase the versatility of UniSparse by supporting the recently proposed Serpens format [Song et al. 2022a], which enables high-performance sparse processing on FPGAs (§7.2.3). Furthermore, we demonstrate that UniSparse can handle partitioned formats for PIM systems using C^2SR and CISR. Finally, we explore a more load-balanced version of CISR, referred to as CISR-plus, by leveraging the custom map functions and the `Reorder` primitive in UniSparse (§7.2.4).

7.2.1 Case Study: The Hybrid BDIA/CSR Format on CPUs. We generate the hybrid BDIA/CSR format using UniSparse. Initially, the input matrix is decomposed into two sub-matrices, both represented in COO format. One sub-matrix is converted to the BDIA format, while the other is converted to the CSR format. The decomposition pattern is adjusted by tuning the blocking factor and the non-zero threshold of each diagonal, which can affect the performance of the sparse kernel. In this work, we manually select the blocking size and thresholds with the best performance. Automatically searching for the optimal decomposition factors will be left for future work. The block sizes and thresholds used for each dataset are presented above the result bars on the right side of Figure 10.

We profile the execution time breakdown of converting COO to BDIA on multiple datasets and summarize the results in the left side of Figure 10. The `Sort` operator is critical and accounts for 65% of the total conversion time since it involves intensive memory reads and writes.

As shown in the right side of Figure 10, we evaluate the SpMV kernel with the hybrid BDIA/CSR format (program in Figure 3d) in single precision on four hardware configurations, using a 48-core Intel Xeon Gold 6248R CPU at 3.00 GHz and an NVIDIA RTX A6000 GPU. The SpMV kernel on the CPU is parallelized using OpenMP, and the kernel on the GPU leverages APIs provided by the cuSPARSE library. The heterogeneous configuration runs the SpMV kernel with one sub-matrix in BDIA on the CPU, and the other sub-matrix in CSR on the GPU simultaneously. The homogeneous configuration runs SpMV with both sub-matrices in BDIA and CSR formats on the CPU. Another two configurations run the SpMV kernel with only the CSR format implemented in cuSPARSE on the GPU, and in the Intel MKL library on the CPU. For the MKL implementation of SpMV, we store the matrix in the CSR format (`mkl_sparse_s_create_csr`), called the optimization function (`mkl_sparse_optimize`) and the Inspector-Executor (IE) routine (`mkl_sparse_s_mv`). The heterogeneous configuration yields 5.63×, 1.28×, and 10.73× speedup in Geomean over the SpMV kernel with the hybrid format only on the CPU (homogeneous configuration), using cuSPARSE on the GPU, and using MKL library on the CPU, respectively.

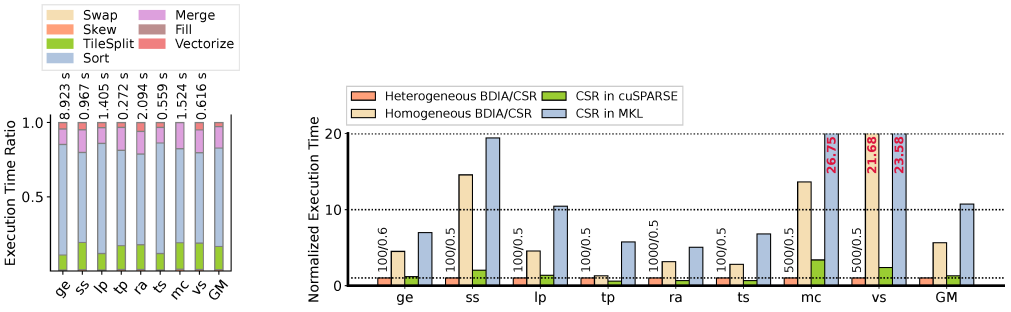


Fig. 10. The case study of the hybrid BDIA/CSR format. The left figure shows the breakdown of the execution time for converting from COO to BDIA format. The right figure shows the performance comparison of the SpMV kernel across various configurations. The execution times are normalized to the heterogeneous configuration, and execution times over 20 are marked with red numbers inside bars.

7.2.2 Case Study: The Hybrid BELL/COO Format on GPUs. The hybrid BELL/COO format is generated by decomposing the input matrix into two sub-matrices, one converting to the BELL format while the other remaining in the COO format. The time breakdown of the conversion from COO to BELL is profiled across multiple datasets, and the results are summarized on the left side of Figure 11. We manually tune the decomposition parameters by adjusting the block size and the non-zero threshold within each block, selecting the configuration that yields the best performance. These decomposition parameters are indicated above each result bar on the right side of Figure 11.

We evaluate sparse matrix-matrix multiplication (SpMM) in single precision using the hybrid BELL/COO format and compare it with the one using only the CSR format. The compute kernel is deployed on an NVIDIA RTX A6000 GPU through APIs provided by the cuSPARSE library. Figure 11 shows the normalized run time of the SpMM kernel using the CSR format vs. the BELL/COO format. The hybrid BELL/COO format on the selected sparse matrices leads to a 2.7× Geomean speedup.

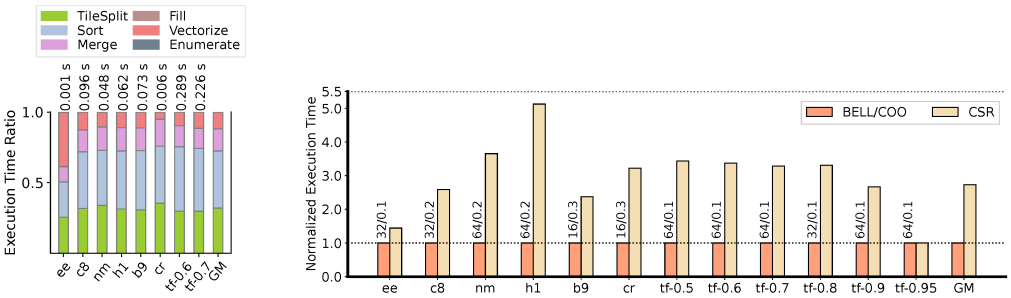
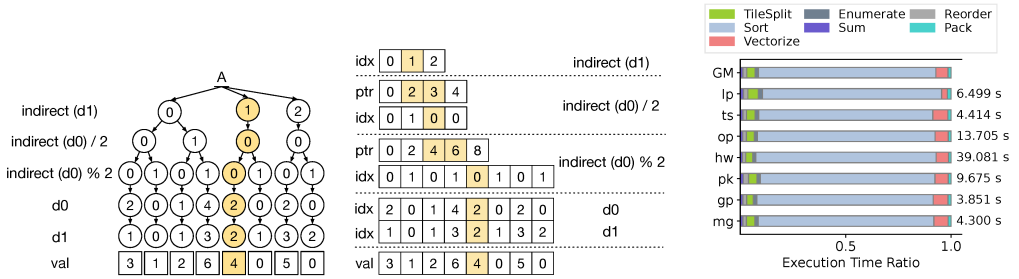


Fig. 11. The case study of the hybrid BELL/COO format. The left figure shows the breakdown of execution time for converting from COO to the BELL format. The right figure shows the normalized run time of cuSPARSE SpMM using the BELL/COO decomposed by UniSparse vs. CSR on NVIDIA A6000. Datasets transformer-50 to 95 are pruned weight matrices of Transformer [Gale et al. 2019] with sparsity ranging from 50% to 95%.

7.2.3 Case Study: The Serpens Format on FPGAs. We evaluate the sparse format proposed in the Serpens accelerator [Song et al. 2022a,b] and demonstrate how UniSparse can express and generate the Serpens format using indirect functions and query primitives that change the order of elements. As shown in Figures 12a and 12b, the Serpens format traverses elements in column order, preserving the dependency length between adjacent row elements with a few padding zeros. This allows the Serpens accelerator to achieve significant throughput improvement for the SpMV kernel.

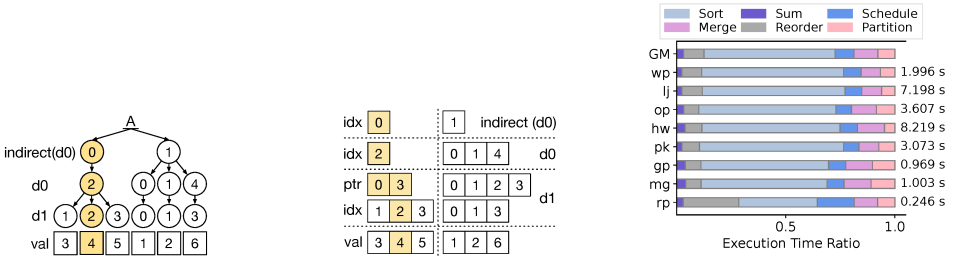
UniSparse automatically converts from COO to the Serpens format. Despite the `Sort` operator consuming a significant portion (84% in Geomean) of the total time (Figure 12c), UniSparse is able to generate this high-performance format in seconds, significantly improving the productivity of hardware developers and providing easier access to sparse acceleration for software developers. According to [Song et al. 2022a], utilizing this custom format in Serpens results in 1.91× better throughput and 1.71× better energy efficiency on an AMD Xilinx Alveo U280 device compared to the prior state-of-the-art FPGA accelerator [Hu et al. 2021].

7.2.4 Case Study: The C^{2}SR, CISR, and CISR-plus Formats on PIMs. We evaluate SpMV using the C^{2}SR and CISR formats on a simulated PIM device [Devic et al. 2022] with 1024 and 2048 cores. Each PIM core has a copy of the input dense vector and computes a subset of the output vector in a lock-free execution pattern. Figure 14a shows the maximum vs. the average number of non-zeros processed per core. As the number of cores increases, the load imbalance introduced by the C^{2}SR format gradually becomes a bottleneck, whereas the CISR format mitigates this issue. Figure 14b



(a) The data structure of Serpens (b) The data layout of Serpens (c) Breakdown of COO→Serpens

Fig. 12. The Serpens format of the matrix A in Figure 1a.



(a) The data structure of CISR-plus (b) The data layout of CISR-plus (c) Breakdown of COO→CISR-plus

```
1  #CISR-plus = #encoding<{ idx_map<(d0,d1)->(indirect(d0),d0,d1)>, mutation<merge(0,1), trim(1,2)>,
2    indirect<{ sum(value) groupBy (d0,d1)->(d0) with value ne 0 -> 1 | otherwise -> 0
3               reorder(d0) traverseBy (d0,d1)->(d0)
4               schedule(d0) traverseBy (d0,d1)->(d0/2) }>,
5    layout<partition(0)> }>
```

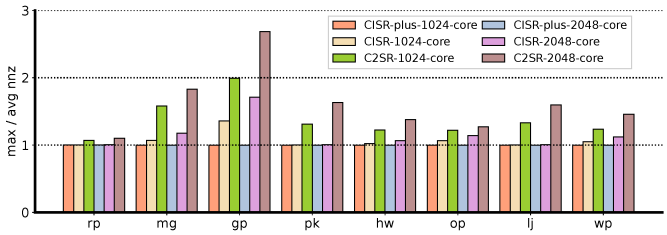
(d) The format encoding of CISR-plus.

Fig. 13. The CISR-plus format of the matrix A in Figure 1a.

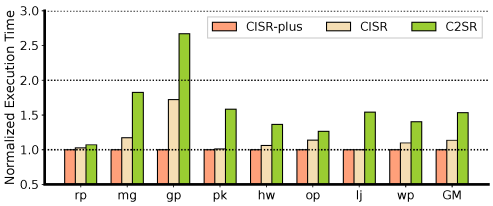
shows the normalized execution time of SpMV on 1024 PIM cores using the C^{2}SR vs. the CISR format. Compared with the C^{2}SR format, using the CISR format improves the performance by 1.35× in Geomean.

Using UniSparse, we can further explore an optimized version of CISR, called CISR-plus. CISR-plus schedules rows with the maximum number of non-zeros first, which leads to better load balance among PIM cores. The data structure and layout of CISR-plus are shown in Figures 13a and 13b. The **`reorder`** primitive along with custom map functions enable us to express (Figure 13d) and generate the CISR-plus format using UniSparse. The conversion time breakdown is shown in Figure 13c. We can see that the `Sort` operator still dominates the conversion time, while the `Reorder` and `Schedule` operators are relatively faster. The improved load balance by using CISR-plus format is demonstrated in Figure 14a. We further evaluate the performance of CISR-plus by comparing it to CISR and C^{2}SR when running SpMV on 1024 PIM cores, as shown in Figure 14b. Using CISR-plus leads to a 1.14× and 1.54× speedup in Geomean over CISR and C^{2}SR, respectively.

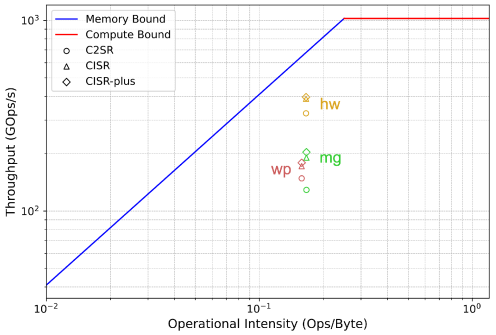
Figure 14c presents a roofline analysis of employing C^{2}SR, CISR, and CISR-plus formats for 3 selected matrices using 1024 PIM cores. While all kernels are memory-bound, adopting the CISR-plus format brings the design nearer to the roofline by addressing the load imbalance issue among PIM cores, thereby increasing throughput.



(a) Load imbalance on different numbers of PIM cores.



(b) Normalized run time on 1024 PIM cores.



(c) Roofline analysis for 3 selected matrices on 1024 PIM cores.

Fig. 14. The SpMV profiling results using CISR-plus, CISR, and C^{2}SR formats on the simulated PIM device.

7.2.5 Case Study: Label Propagation on GPUs. We evaluate the label propagation application using custom formats generated by UniSparse and compare it against the traditional CSR format. Label propagation is a semi-supervised machine learning algorithm that iteratively propagates label information from previously labeled data points to unlabeled ones [Iscen et al. 2019]. This algorithm operates through iterative matrix multiplications, where the right matrix, referred to as the label matrix, contains the label vectors for each data point (or node), and the left one is the adjacency matrix that represents the node connectivity. Initially, the label matrix is sparse, with only a few nodes having non-zero values in each label category. While employing sparse-sparse matrix multiplication (SpGEMM) for the initial stage of the algorithm may lead to better performance, as the density of the label matrix increases, it requires ad hoc experiments for each dataset to determine the density threshold to switch using SpMM for the rest of the computation. In our experiments, for simplicity, we use SpMM for the entire process of label propagation.

Table 4 presents the evaluation results of the label propagation algorithm. The label propagation is implemented through 20 iterations of SpMM. Before computation, we normalize the sparse matrices to ensure each row sums up to 1, and we generate label matrices with a fixed column size of 1000. The hybrid BELL/COO format is generated by UniSparse using 24 threads on an Intel Xeon Gold 6242 CPU at 2.80 GHz. The compute kernels are executed on an NVIDIA RTX A6000 GPU through APIs provided by the cuSPARSE library.

As presented in Table 4, the total execution time of the label propagation algorithm using the hybrid BELL/COO format includes a format preprocessing time and a kernel execution time. The preprocessing of the hybrid BELL/COO format involves intensive memory accesses to perform both matrix decomposition and format conversion, thus introducing a non-negligible overhead to overall performance. However, in applications involving multiple iterations of sparse linear algebra kernels, such as label propagation, the overhead from format preprocessing is amortized across

iterations. This allows for an overall speedup by utilizing custom sparse formats. Our evaluation shows that using the hybrid BELL/COO format for the label propagation application leads to a 1.61× speedup in Geomean compared to the traditional CSR format.

Table 4. The case study of label propagation application. The numbers for each execution time (in *milliseconds*) ± standard deviation are computed based on 20 independent runs.

Data	CSR Format	Hybrid BELL/COO Format			Speedup over CSR
		Format Preprocessing	Kernel Execution	Total	
ee	11.969 ± 0.106	5.490 ± 0.644	4.460 ± 0.136	9.950 ± 0.658	1.20
c8	190.272 ± 1.149	63.022 ± 2.257	55.960 ± 0.246	118.983 ± 2.270	1.60
nm	96.842 ± 0.666	21.376 ± 2.665	49.809 ± 0.119	71.185 ± 2.667	1.36
h1	111.431 ± 0.589	27.172 ± 1.509	16.163 ± 0.125	43.335 ± 1.514	2.57
cr	11.764 ± 0.101	3.884 ± 0.146	3.661 ± 0.006	7.545 ± 0.146	1.56
my	32.629 ± 0.015	15.209 ± 0.555	11.162 ± 0.528	26.371 ± 0.766	1.24
od	246.337 ± 0.280	69.728 ± 3.687	55.450 ± 0.204	125.177 ± 3.693	1.97
wv	1464.949 ± 7.565	380.897 ± 10.194	458.716 ± 0.415	839.613 ± 10.202	1.74
Geomean					1.61

7.3 Format Conversion

Compared to state-of-the-art compilers, UniSparse offers a wider coverage of supported formats in its automatic format conversion routines. Table 5 lists 7 representative format conversion cases supported by UniSparse. In the listed format conversion cases, both MLIR SparseTensor dialect [Bik et al. 2022] and TACO [Chou et al. 2020a] lack stable support for DCSC → BCSR, CSB → DIA-variant, COO → C^2SR, and COO → CISR, while MLIR SparseTensor does not implement COO → DIA nor COO → ELL. Additionally, to the best of our knowledge, none of the prior sparse linear algebra compilers support specialized formats like CISR and Serpens.

Furthermore, UniSparse demonstrates comparable performance in the given format conversion cases compared to two baseline compilers. Format conversion is memory-intensive, and both UniSparse and TACO optimize their code to minimize memory accesses. However, differences in the implementations of code generation lead to performance variations. The format decoding in UniSparse introduces performance overhead, which leads to inferior performance for CSR → CSC and COO → ELL conversions compared with TACO on small matrices (`ch`, `mj`, `sh`, `bm`) with high sparsity (~1.0e-5). We also notice that the transpose operation in MLIR SparseTensor utilizes an auxiliary class to enumerate values in any permutation order of tensor dimensions. While this approach offers functional versatility, it leads to increased time complexity and more memory accesses, resulting in slower performance when converting CSR to CSC.

7.4 Compute Kernel Generation

Table 6 provides a performance comparison of SpMM and SpGEMM kernels across various formats generated by UniSparse, MLIR SparseTensor, and TACO. The SpMM kernel performs matrix multiplication between a sparse matrix in the dataset and a synthetic dense matrix with a fixed column size of 1000. The SpGEMM kernel conducts matrix multiplication between a sparse matrix in the dataset and itself. These kernels operate with double precision and execute in a single-threaded environment on an Intel Xeon Gold 6248R CPU at 3.00 GHz.

UniSparse leverages the kernel generation passes from the MLIR SparseTensor dialect for compute kernels including CSR SpMM, DCSC SpMM, CSR×CSR→CSR SpGEMM, and CSC×CSC→CSC SpGEMM. However, UniSparse and MLIR SparseTensor differ in the implementation of the tensor storage. The tensor initialization function of MLIR SparseTensor is slightly more complicated.

Table 5. Comparison of the actual execution times for format conversion programs generated by UniSparse, the MLIR SparseTensor dialect under the LLVM 15.0.0 release, and the format conversion artifact of TACO [Chou et al. 2020b]. The numbers for each execution time (in *seconds*) ± standard deviation are computed based on 20 independent runs.

Data	CSR → CSC			DCSC → BCSR	CSB → DIA-variant
	TACO	SparseTensor	UniSparse	UniSparse	UniSparse
wv	0.001 ± 0.000	0.003 ± 0.000	0.001 ± 0.000	0.012 ± 0.001	0.067 ± 0.001
ee	0.000 ± 0.000	0.001 ± 0.000	0.000 ± 0.000	0.003 ± 0.000	0.004 ± 0.000
nm	0.005 ± 0.000	0.031 ± 0.001	0.002 ± 0.000	0.039 ± 0.000	0.032 ± 0.001
cm	0.001 ± 0.000	0.006 ± 0.000	0.001 ± 0.000	0.012 ± 0.001	0.007 ± 0.000
ct	0.018 ± 0.000	0.078 ± 0.004	0.009 ± 0.000	0.148 ± 0.001	0.184 ± 0.002
lp	0.191 ± 0.004	0.659 ± 0.005	0.193 ± 0.003	2.554 ± 0.015	4.509 ± 0.031
tp	0.168 ± 0.004	0.562 ± 0.004	0.139 ± 0.002	2.249 ± 0.010	6.157 ± 0.038
ts	0.162 ± 0.002	0.462 ± 0.006	0.168 ± 0.001	1.733 ± 0.009	2.509 ± 0.017
ch	0.001 ± 0.000	0.006 ± 0.000	0.001 ± 0.000	0.015 ± 0.001	0.009 ± 0.000
mj	0.006 ± 0.000	0.043 ± 0.001	0.008 ± 0.000	0.134 ± 0.001	0.178 ± 0.002
sh	0.001 ± 0.000	0.009 ± 0.001	0.001 ± 0.000	0.023 ± 0.000	0.018 ± 0.000
bm	0.002 ± 0.000	0.018 ± 0.000	0.003 ± 0.000	0.058 ± 0.001	0.063 ± 0.001
Geomean Speedup	1	0.20	1.13	1	1

Data	COO → ELL		COO → DIA		COO → C²SR	COO → CISR
	TACO	UniSparse	TACO	UniSparse	UniSparse	UniSparse
wv	0.018 ± 0.000	0.014 ± 0.001	0.092 ± 0.001	0.065 ± 0.004	0.003 ± 0.000	0.003 ± 0.000
ee	0.002 ± 0.000	0.001 ± 0.000	0.005 ± 0.000	0.003 ± 0.000	0.001 ± 0.000	0.001 ± 0.000
nm	0.011 ± 0.000	0.006 ± 0.000	0.008 ± 0.000	0.003 ± 0.000	0.017 ± 0.000	0.017 ± 0.000
cm	0.002 ± 0.000	0.002 ± 0.000	0.001 ± 0.000	0.001 ± 0.000	0.005 ± 0.000	0.005 ± 0.000
ct	0.048 ± 0.000	0.023 ± 0.000	0.031 ± 0.000	0.013 ± 0.000	0.071 ± 0.001	0.077 ± 0.001
lp	0.289 ± 0.006	0.298 ± 0.006	0.433 ± 0.004	0.352 ± 0.008	1.069 ± 0.004	1.080 ± 0.021
tp	0.208 ± 0.005	0.220 ± 0.002	0.133 ± 0.003	0.127 ± 0.002	0.929 ± 0.009	1.303 ± 0.062
ts	0.274 ± 0.003	0.249 ± 0.004	0.192 ± 0.002	0.163 ± 0.002	0.541 ± 0.005	0.508 ± 0.045
ch	0.001 ± 0.000	0.002 ± 0.000	0.001 ± 0.000	0.001 ± 0.000	0.006 ± 0.001	0.007 ± 0.001
mj	0.010 ± 0.000	0.017 ± 0.000	0.010 ± 0.000	0.008 ± 0.000	0.060 ± 0.001	0.072 ± 0.001
sh	0.002 ± 0.000	0.003 ± 0.000	0.002 ± 0.000	0.001 ± 0.000	0.011 ± 0.000	0.013 ± 0.000
bm	0.004 ± 0.000	0.007 ± 0.000	0.003 ± 0.000	0.004 ± 0.000	0.026 ± 0.000	0.031 ± 0.000
Geomean Speedup	1	1.01	1	1.37	1	1

Therefore, the compute kernels generated by MLIR SparseTensor are slightly slower in some cases but mostly on par with UniSparse.

The compute kernels generated by TACO show slower performance due to its different kernel generation strategies. For instance, in the case of CSR×CSR→CSR SpGEMM, all three compilers generate the row-wise product implementation, while the TACO-generated kernel involves an additional sorting of a partial sum index buffer, which makes it slower. We exclude the TACO column for the CSC×CSC→CSC SpGEMM kernel because the kernel generated by TACO produces incorrect outputs according to our experiments.

Furthermore, UniSparse generates the SpMM kernel with the DIA-variant format using the kernel generation method described in Section 5.3, whereas MLIR SparseTensor and TACO lack support for this format. From Table 6, we observe that the performance of SpMM using the DIA-variant format is inferior compared to the CSR or DCSC format on the common datasets `ee`, `ch`, `sh`, `mj`, and `bm`. This performance gap is primarily due to the extra computation required for zero paddings along the diagonals in the DIA-variant format during single-threaded execution. As presented in Section 7.2.1 and 7.2.2, formats with zero paddings exhibit better performance in a multi-threaded setting where computation among the padded dimensions can be parallelized.

Table 6. Comparison of the actual execution times for compute kernels generated by UniSparse, the MLIR SparseTensor dialect under the LLVM 15.0.0 release, and the main branch of TACO [Kjolstad et al. 2019]. The numbers for each execution time (in *seconds*) ± standard deviation are computed based on 20 independent runs.

Data	CSR SpMM			DCSC SpMM			Data	DIA-variant SpMM
	TACO	SparseTensor	UniSparse	TACO	SparseTensor	UniSparse		UniSparse
m2	9.682 ± 0.109	1.937 ± 0.309	1.847 ± 0.038	9.608 ± 0.059	1.889 ± 0.086	1.857 ± 0.018	bw	0.078 ± 0.001
sc	4.344 ± 0.046	0.727 ± 0.005	0.734 ± 0.042	4.410 ± 0.314	0.739 ± 0.034	0.730 ± 0.007	af	8.136 ± 0.135
pg	0.877 ± 0.009	0.154 ± 0.002	0.153 ± 0.006	0.875 ± 0.011	0.151 ± 0.006	0.150 ± 0.002	cg	0.528 ± 0.012
ca	8.162 ± 0.495	1.534 ± 0.020	1.498 ± 0.030	7.983 ± 0.080	1.567 ± 0.062	1.528 ± 0.027	ex	22.642 ± 0.323
f3	10.327 ± 0.137	1.749 ± 0.139	1.702 ± 0.059	10.247 ± 0.079	1.765 ± 0.055	1.729 ± 0.038	cm	2.026 ± 0.027
cc	0.818 ± 0.011	0.185 ± 0.004	0.187 ± 0.009	0.813 ± 0.008	0.188 ± 0.006	0.187 ± 0.003	ct	32.896 ± 0.460
p3	1.364 ± 0.020	0.298 ± 0.007	0.302 ± 0.021	1.353 ± 0.013	0.302 ± 0.006	0.302 ± 0.006	nm	7.841 ± 0.133
ee	0.098 ± 0.002	0.014 ± 0.001	0.014 ± 0.001	0.098 ± 0.002	0.014 ± 0.001	0.014 ± 0.001	ee	3.806 ± 0.044
ch	0.918 ± 0.007	0.132 ± 0.007	0.136 ± 0.009	0.917 ± 0.007	0.136 ± 0.008	0.135 ± 0.008	ch	2.179 ± 0.084
sh	1.561 ± 0.008	0.223 ± 0.015	0.223 ± 0.016	1.566 ± 0.012	0.230 ± 0.015	0.222 ± 0.002	sh	5.821 ± 0.202
mj	6.997 ± 0.049	1.165 ± 0.196	1.133 ± 0.124	6.993 ± 0.051	1.101 ± 0.026	1.125 ± 0.024	mj	30.623 ± 0.472
bm	3.266 ± 0.011	0.533 ± 0.031	0.542 ± 0.049	3.274 ± 0.024	0.546 ± 0.043	0.529 ± 0.005	bm	8.672 ± 0.235
Geomean Speedup	1	5.77	5.81	1	5.72	5.81		1

Data	CSR×CSR→CSR SpGEMM			CSC×CSC→CSC SpGEMM	
	TACO	SparseTensor	UniSparse	SparseTensor	UniSparse
m2	0.610 ± 0.010	0.390 ± 0.004	0.305 ± 0.006	0.388 ± 0.005	0.309 ± 0.006
sc	0.434 ± 0.003	0.245 ± 0.002	0.242 ± 0.003	0.244 ± 0.003	0.244 ± 0.008
pg	0.050 ± 0.002	0.032 ± 0.001	0.028 ± 0.001	0.031 ± 0.001	0.025 ± 0.001
ca	1.384 ± 0.010	0.725 ± 0.007	0.657 ± 0.014	0.720 ± 0.012	0.654 ± 0.006
f3	2.041 ± 0.017	1.178 ± 0.017	1.058 ± 0.008	1.166 ± 0.010	1.062 ± 0.008
cc	0.268 ± 0.004	0.146 ± 0.002	0.137 ± 0.001	0.143 ± 0.002	0.137 ± 0.002
p3	0.385 ± 0.005	0.191 ± 0.003	0.179 ± 0.003	0.189 ± 0.002	0.178 ± 0.002
ee	0.055 ± 0.005	0.021 ± 0.002	0.020 ± 0.001	0.021 ± 0.001	0.019 ± 0.001
ch	0.034 ± 0.001	0.012 ± 0.001	0.012 ± 0.001	0.012 ± 0.000	0.013 ± 0.000
sh	0.047 ± 0.001	0.020 ± 0.002	0.020 ± 0.001	0.019 ± 0.000	0.021 ± 0.004
mj	0.504 ± 0.006	0.251 ± 0.022	0.219 ± 0.015	0.242 ± 0.013	0.217 ± 0.001
bm	0.154 ± 0.001	0.079 ± 0.006	0.079 ± 0.007	0.077 ± 0.004	0.077 ± 0.005
Geomean Speedup	1	1.98	2.15	1	1.06

8 CONCLUSION

We present UniSparse, an intermediate language that describes sparse tensor formats using decoupled data structures and data layouts. Our approach formulates the data structures of a sparse format virtually as a metadata tree, described using an index map and a set of structure mutation primitives. To enhance the expressibility of sparse data structures, we introduce query primitives that allow custom index map functions. Data layouts are determined using layout primitives that enable switching from SoA to AoS layout and partitioning a tensor. With the well-formulated format description, the UniSparse compiler automates format customization, as well as code generation for format conversion and compute kernels. Our work facilitates research into efficient formats for various types of tensors, expediting the development of fast sparse tensor algebra in diverse domains, including machine learning, scientific computing, and data analytics.

DATA-AVAILABILITY STATEMENT

The software supporting this paper is maintained publicly on GitHub [Liu et al. 2024a]. The version submitted to the OOPSLA' 24 Artifact Evaluation Committee (AEC) is permanently archived on Zenodo [Liu et al. 2024b].

ACKNOWLEDGMENTS

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, NSF Awards #1909661, #2019306, #2118709 and #2212371, and by AFRL and DARPA under agreement FA8650-18-2-7863.

REFERENCES

Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and verifying sparse matrix codes. *ACM Sigplan Notices* 45, 9 (2010), 249–260. https://doi.org/10.1145/1863543.1863581

Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231. https://doi.org/10.1137/060676489

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Int'l Conf. on High Performance Computing Networking, Storage and Analysis* (2009), 1–11. https://doi.org/10.1145/1654059.1654078

Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler support for sparse tensor computations in MLIR. *ACM Trans. on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–25. https://doi.org/10.1145/3544559

Aart Bik and Harry Wijshoff. 1993. Compilation techniques for sparse matrix computations. *Int'l Symp. on Supercomputing (ICS)* (1993). https://doi.org/10.1145/165939.166023

Ronald F Boisvert, Ronald F Boisvert, and Karin A Remington. 1996. *The matrix market exchange formats: Initial design.* Vol. 5935. US Department of Commerce, National Institute of Standards and Technology.

Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. *Int'l Parallel and Distributed Processing Symp. (IPDPS)* (2008). https://doi.org/10.1109/IPDPS.2008.4536313

Aydın Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. *Symp. on Parallel Algorithms and Architectures (SPAA)* (2009), 233–244. https://doi.org/10.1145/1583991.1584053

Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM SIGPLAN Notices* 45, 5 (2010), 115–126. https://doi.org/10.1145/1837853.1693471

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30. https://doi.org/10.1145/3276493

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020a. Automatic generation of efficient sparse tensor format conversion routines. (2020), 823–838. https://doi.org/10.1145/3385412.3385963

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020b. *Automatic generation of efficient sparse tensor format conversion routines.* https://github.com/stephenchouca/taco/tree/pldi20ae

Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. on Mathematical Software (TOMS)* 38, 1 (2011). https://doi.org/10.1145/2049662.2049663

Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. *Int'l Symp. on Computer Architecture (ISCA)* (2022). https://doi.org/10.1145/3470496.3527431

Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. 2014. A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)* (2014). https://doi.org/10.1109/FCCM.2014.23

Takeshi Fukaya, Koki Ishida, Akie Miura, Takeshi Iwashita, and Hiroshi Nakashima. 2021. Accelerating the SpMV kernel on standard CPUs by exploiting the partially diagonal structures. *arXiv preprint arXiv:2105.04937* (2021).

Trevor Gale, Erich Elsen, and Sara Hooker. 2019. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574* (2019).

Dahai Guo, William Gropp, and Luke N Olson. 2016. A Hybrid Format for Better Performance of Sparse Matrix-Vector Multiplication on a GPU. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 103–120. https://doi.org/10.1177/1094342015593156

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Advances in Neural Information Processing Systems*

(2020). https://doi.org/10.5555/3495724.3497579

Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)* (2021). https://doi.org/10.1109/ICCAD51958.2021.9643582

Eun-Jin Im and Katherine Yelick. 1998. Model-based memory hierarchy optimizations for sparse matrices. *Workshop on Profile and Feedback-Directed Compilation* 139 (1998).

Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondrej Chum. 2019. Label propagation for deep semi-supervised learning. *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2019), 5070–5079. https://doi.org/10.1109/CVPR.2019.00521

Robert Johansson and Robert Johansson. 2015. Sparse Matrices and Graphs. *Numerical Python: A Practical Techniques Approach for Industry* (2015), 235–254. https://doi.org/10.1007/978-1-4842-0553-2_10

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

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. https://doi.org/10.1145/3133901

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2019. *The Tensor Algebra Compiler.* https://github.com/tensor-compiler/taco

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. *Compiling parallel sparse code for user-defined data structures.* Technical Report. Cornell University.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014). https://doi.org/10.1145/2898361

Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. 2024a. *UniSparse: An Intermediate Language for General Sparse Format Customization.* Cornell University. https://github.com/cornell-zhang/UniSparse

Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. 2024b. *UniSparse: An Intermediate Language for General Sparse Format Customization.* Cornell University. https://doi.org/10.5281/zenodo.10464500

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

Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003

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

Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022a. Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication. *Design Automation Conf. (DAC)* (2022), 211–216. https://doi.org/10.1145/3489517.3530420

Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022b. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. (2022), 65–77. https://doi.org/10.1145/3490422.3502357

Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. *Int'l Symp. on Microarchitecture (MICRO)* (2020). https://doi.org/10.1109/MICRO50266.2020.00068

Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. (12 2021). https://doi.org/10.1109/LLVMHPC54804.2021.00009

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* 3 (2023), 660–678. https://doi.org/10.1145/3582016.3582047

Received 21-OCT-2023; accepted 2024-02-24