An Intermediate Language for General Sparse Format Customization

Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang, *Fellow, IEEE*

***Abstract*—The inevitable trend of hardware specialization drives an increasing use of custom data formats in processing sparse workloads, which are typically memory-bound. These formats facilitate the automated generation of target-aware data layouts to improve 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. Moreover, since these frameworks adopt an attribute-based approach for format abstraction, they cannot easily be extended to support general format customization. To overcome this deficiency, we propose UniSparse, an intermediate language that provides a unified abstraction for representing and customizing sparse formats. We also develop a compiler leveraging the MLIR infrastructure, which supports adaptive customization of formats. We demonstrate the efficacy of our approach through experiments running commonly-used sparse linear algebra operations with hybrid formats on multiple different hardware targets, including an Intel CPU, an NVIDIA GPU, and a simulated processing-in-memory (PIM) device.**

***Index Terms*—Compilers, heterogeneous (hybrid) systems, specialized application languages, sparse linear algebra.**

I. Introduction

As Dennards 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. 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. 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. A sparse tensor is commonly represented in a specialized data structure, also known as a *sparse format*, which exploits the sparsity of the tensor to reduce storage size and memory footprint. These sparse formats only store the non-zero elements/blocks of the tensor, along with the associated coordinates. These coordinates are encoded in a compressed form as *metadata*. Conceptually, the metadata can be viewed as a tree that captures the multi-dimensional coordinates hierarchically; thus it requires multiple indirect memory accesses to "walk" the tree in order to reconstruct the original coordinates of a non-zero element. Due to the extensive use of such data structures, sparse workloads typically exhibit irregular and input-dependent compute and data access patterns, which make them memory-bound.

To improve 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 [1], [4], [12] and banked formats for FPGAs [9], [14] and dedicated accelerators [20]. 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. 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 the custom formats needs to account for input-dependent sparsity patterns, inherent parallelism of the dominant compute kernels and the target hardware.

Prior research has attempted to address the productivity challenge. Sparse linear/tensor algebra libraries (e.g., sparse BLAS, Intel MKL, NVIDIA cuSPARSE) achieve high performance, but they only support a limited set of sparse formats. Recent efforts on sparse tensor algebra compilers such as TACO [5], [15], COMET [21], and SparseTIR [22] describe tensor dimensions in attributes (e.g., `sparse` or `dense`), and generate sparse tensor algebra kernels assisted by pre-defined code generation templates. This attribute-based format abstraction limits their extensibility to custom formats, with memory access patterns and index-matching schemes dictated by the code generation templates.

This work proposes UniSparse, the first intermediate language for general sparse format customization, which can (1) systematically express an unlimited number of custom formats, (2) support format customization with the awareness of input sparsity, compute operations, and hardware targets, and (3) for the new formats, automate code generation of their compute operations and conversion with other formats. We observe that the storage layout of a sparse tensor can generally be viewed as a map from the non-zeros' coordinates to a tree of indices. More importantly, this map can be expressed using a few queries and storage mutations, which we call primitives. These primitives are the key ingredients of a concise, language-based abstraction for specifying custom sparse formats, including but not limited to many previously proposed high-performance formats. We implement the compiler on top of a multi-level compiler infrastructure, namely MLIR [17]. Our automation flow

Manuscript received 9 February 2023; revised 4 March 2023; accepted 24 March 2023. Date of publication 28 March 2023; date of current version 19 October 2023. 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 under Grants 1909661, 2019306, 2118709, and 2212371, and in part by AFRL and DARPA under Grant FA8650-18-2-7863. *(Corresponding author: Jie Liu.)*

Jie Liu, Zhongyuan Zhao, and Zhiru Zhang are with the School of ECE, Cornell University, Ithaca, NY 14850 USA (e-mail: jl3952@cornell.edu; zz546@cornell.edu; zhiruz@cornell.edu).

Zijian Ding was with Peking University, Beijing 100871, China. He is now with the University of California Los Angeles, Los Angeles, CA 900951 USA (e-mail: bradyd@cs.ucla.edu).

Benjamin Brock and Hongbo Rong are with the Intel, Santa Clara, CA 95054 USA (e-mail: benjamin.brock@intel.com; hongbo.rong@intel.com).

Digital Object Identifier 10.1109/LCA.2023.3262610

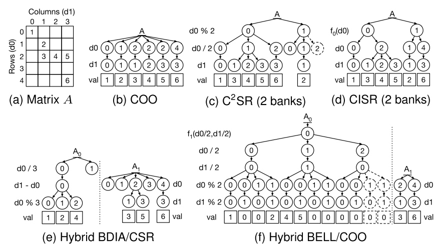


Fig. 1. Selected custom formats of a tensor (here the matrix A). The root of a metadata tree is the symbol of the tensor. A_0 and A_1 are sub-matrices of the matrix A.

TABLE I
STATE-OF-THE-ART SPARSE TENSOR ALGEBRA COMPILERS

	Prior Work	Input-awareness	Hardware-awareness	Auto format Conversion
Attribute-based	TACO [5], [15]	○	◒	○
	MLIR SparseTensor [2]	○	◒	◒
	COMET [22]	○	◒	○
	SparseTIR [23]	◒	◒	◒
Language-based	TACO-conversion [6]	○	◒	●
	UniSparse	●	●	●

demonstrates significant productivity improvement and high performance on multiple hardware backends including CPUs, GPUs, and a simulated PIM device [8].

II. Background and Motivation

In this section, we discuss several high-performance sparse formats and prior work on sparse format abstraction.

A. Custom Sparse Formats

A sparse format consists of two parts: element values and metadata. The metadata captures the correspondence between the elements and their coordinates, typically in a compressed form. While different formats have different ways to implement the metadata, it is conceptually organized in a tree structure, where each path from the root to a leaf node indexes an element.

Fig. 1 illustrates several sparse formats of the matrix A in Fig. 1(a). Fig. 1(b) shows the traditional coordinate (COO) format. Fig. 1(c) and (d) illustrate two hardware-friendly formats: the cyclic channel sparse row (C^2SR) format [20] interleaves the rows of a tensor into sub-tensors, one sub-tensor for one memory bank, so as to increase memory bandwidth utilization; the compressed interleaved sparse row (CISR) format [9] distributes rows to memory channels in a load balancing manner. Fig. 1(e) and (f) show the hybrid blocked-diagonal (BDIA)/CSR [10] and the hybrid blocked-ELLPACK (BELL)/COO format [1], [12]. The BDIA format partitions the rows of a tensor and stores partial diagonals with their offsets. The BELL format stores the same number of non-zero blocks at row dimension [4]. By customizing the layout for sub-tensors with different sparsity patterns, hybrid formats can offer higher performance than single formats [1], [10] .

B. Prior Work on Sparse Format Abstraction

Early research on sparse tensor algebra compilers [3], [18] generates compute kernels with hard-coded storage formats. In Table I, we summarize recent work on sparse tensor algebra compilers with format abstraction and categorize them into two classes.

Attribute-Based Abstraction: Prior work such as TACO [5], [15], MLIR's SparseTensor dialect [2], COMET [21] and SparseTIR [22] describe formats using per-dimension attributes to determine the iteration and access patterns for code generation. However, the attribute-based format abstraction does not scale to express novel formats, as an increasing amount of attributes and lowering routines will be introduced. MLIR's SparseTensor dialect and SparseTIR leverage index maps to improve the expressibility; but the limited combination of axis attributes still prevents them from supporting an unlimited number of new possible formats. Moreover, it is difficult to support fully automated format conversion in these efforts.

Language-Based Abstraction: Recent work [6] describes sparse formats using a language-based approach to assist format conversion, but the language is not well-defined as a format abstraction, and many specialized formats such as hybrid formats are not supported. UniSparse proposes a language-based format abstraction, which complements prior approaches on format customization.

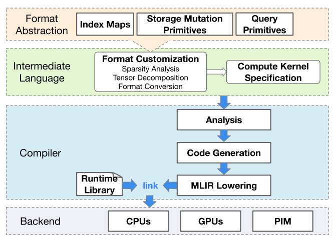


Fig. 2. An overview of the automated compiler flow.

III. UniSparse: A Unified Abstraction for Customizing Sparse Data Formats

We propose UniSparse, an intermediate language that can formally and concisely describe various sparse formats, facilitating both format conversion and customization (Section III-A). Fig. 2 outlines the overall compilation flow. In our approach, a sparse tensor format is encoded as an index map and a set of storage mutation and query primitives, decoupled from the compute kernel specification. The compiler decodes how formats are stored internally and generates code for format pre-processing (Section III-B). Then it can interface with the MLIR SparseTensor work to lower compute operations. The intermediate language is implemented as an independent MLIR dialect and lowered to multiple targets, including CPUs, GPUs, and a PIM simulator [8].

A. Format Abstraction

With UniSparse, the format of a sparse tensor is described by an index map and storage mutation primitives, and sparsity patterns can be obtained through queries.

Index Maps: For a sparse tensor, an index map directly determines the storage layout of the coordinate information in the metadata tree. The map takes a list of source indices as inputs and returns a list of destination arithmetic expressions as results. The order of the index expressions dictates the order of storage. A trivial case of an index map is `(d0, d1) -> (d1, d0)`, which represents a column-major matrix layout.

If we revisit the example in Fig. 1(a), where the matrix is indexed by d_0 and d_1, we can find the destination index expressions associated with each level of the metadata tree for different formats in Fig. 1(b)–1(f). For example, the C^2SR format in Fig. 1(c) is expressed by an index map of `(d0, d1) -> (d0%2, d0/2, d1)`, which indicates that the first dimension of the

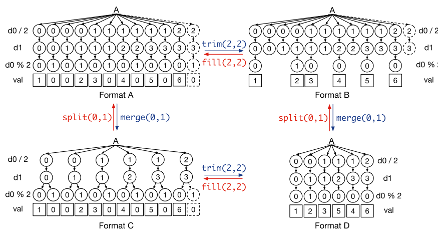


Fig. 3. Examples of storage mutation primitives. Format A, B, C, and D represent four different storage layouts of matrix A in Fig. 1(a).

matrix is divided into two partitions in a cyclic way; the BDIA portion of the hybrid format in Fig. 1(e) (i.e., a sub-matrix on the left) is expressed by an index map of `(d0, d1) -> (d0/3, d1-d0, d0%3)`, which partitions the first dimension before grouping data along the diagonals within each partition.

The destination indices are typically expressed as closed-form functions using basic arithmetic operations (e.g., add/sub, multiply, divide, modulo), which we call *direct maps*. For generality, we further allow user-defined functions, namely *indirect maps*, to be used in an index expression. Functions f_0 for the CISR format in Fig. 1(d) and f_1 for the BELL portion of the hybrid format in Fig. 1(f), are such examples. At a high-level, function f_0 distributes rows in a way to balance the number of non-zeros among a set of sub-tensors, and function f_1 computes a new set of indices by enumerating non-zero blocks along the row dimension.

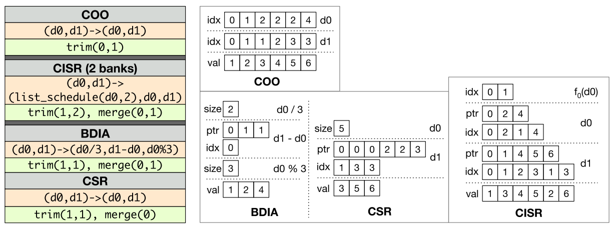
Storage Mutation Primitives: The metadata tree can be further compressed in size using two mutation primitives: **`trim`** and **`merge`**. The **`trim`** primitive prunes away zero values (at the bottom of the tree) and their associated metadata (on the path from the root to the bottom of the tree) between a starting level `S` and an ending level `E`, where `S`$\leq$`E` (i.e., `S` is closer to the root than `E`). The **`merge`** primitive merges equivalent paths at specified levels to save storage space. To support format conversion, we further introduce **`fill`** and **`split`**, which are the reverse of **`trim`** and **`merge`**, respectively. They have exactly the opposite effect on the metadata tree. Fig. 3 illustrates the four primitives using a simple example.

Query Primitives: UniSparse further provides methods to obtain statistics of a sparse tensor through query primitives, each of which consists of a reduction map and an aggregation function.

A reduction map divides tensor elements into groups using the same syntax as an index map. However, the destination expression of a reduction map typically has fewer dimensions than the source expression. For example, the reduction map `(d0, d1) -> (d0%2)` assigns the values in the even rows into one group and the values in the odd rows into the other group. An aggregate function calculates statistics of the groups. We pre-define several aggregation functions. For exmaple, **`count`** returns the number of non-zeros within a group, which is useful to determine sparsity distribution patterns of a tensor.

B. Automation

Analysis: The compiler infers how sparse tensors are stored in memory, i.e., data structures shown in Fig. 4(b), from format abstractions in Fig. 4(a). In general, a metadata tree of a sparse tensor can be stored level by level, and each level by an array of indices (`idx`) and an array of pointers (`ptr`). 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 Fig. 1. In the


COO
`(d0,d1)->(d0,d1)`
`trim(0,1)`
CISR (2 banks)
`(d0,d1)->(list_schedule(d0,2),d0,d1)`
`trim(1,2), merge(0,1)`
BDIA
`(d0,d1)->(d0/3,d1-d0,d0%3)`
`trim(1,1), merge(0,1)`
CSR
`(d0,d1)->(d0,d1)`
`trim(1,1), merge(0)`

(a) Format abstractions (b) Storage data structures

Fig. 4. The abstractions and storage data structures of the selected custom formats of matrix A in Fig. 1.

```
// Format abstraction
#COO = #encoding<{
  indMap=#map<(i,j)->(i,j)>,stgPrim=#prim<trim(0,1)> }>
#CSR = #encoding<{
  indMap=#map<(i,j)->(i,j)>,stgPrim=#prim<merge(0), trim(1,1)> }>
#BDIA = #encoding<{
  indMap=#map<(i,j)->(i div 50, j-i, i mod 50)>,
  stgPrim=#prim<merge(0), trim(1,1)> }>
#COO_COO = #hybrid<{ fmats=[#COO, #COO] }>
#BDIA_CSR = #hybrid<{ fmats=[#BDIA, #CSR] }>

// Format pre-processing
%A1 = unisparse.decompose (%in_A, %thld) {
  rmap=(i,j)->(i div 50, j-i)}: tensor<?x?xf32, #COO_COO>
%A2 = unisparse.convert (%A1): tensor<?x?xf32, #BDIA_CSR>

// Compute operation
#spmv = { indexing_maps = [
  affine_map<(i,j)->(i,j)>, // for argument %A2
  affine_map<(i,j)->(j)>, // for argument %in_X
  affine_map<(i,j)->(i)>], // for argument %out_Y
  iterator_types = ["parallel", "reduction"] }
%0 = linalg.generic #spmv
  ins(%A2, %in_X : tensor<?x?xf32, #BDIA_CSR>, tensor<?xf32>)
  outs(%out_Y: tensor<?xf32>) {
    ^bb0(%a: f32, %b: f32, %x: f32):
      %2 = arith.mulf %a, %b : f32
      %3 = arith.addf %x, %2 : f32
      linalg.yield %3 : f32
  } -> tensor<?xf32>
```

Fig. 5. A UniSparse program for SpMV.

special case that the indices at the current level are contiguous numbers starting from 0, the `idx` array can simply be replaced by a `size`. In another case when nodes have a one-to-one correspondence with their parents. the `ptr` array can be skipped. The **`trim`**`(S,E)` primitive and indirect maps require explicit `idx` arrays at input levels, as indices are no longer continuous after applying these primitives. The **`merge`** primitive adds a `ptr` array to the descendant levels of specified levels.

Code Generation: The compiler lowers the declarative **`convert`** operation into a sequence of mutation primitives (i.e., **`trim/fill`**, **`merge/split`**) and arithmetic operations (e.g., add/sub, multiply, divide, modulo). The mutation primitives are applied to the storage of the source format to make it align with the target format. The arithmetic operations are generated to transform from the source index list to target index expressions.

C. An Illustration of the Intermediate Language

Fig. 5 illustrates UniSparse with sparse matrix-vector multiplication (SpMV) described in MLIR. The original format of the input tensor, COO, and the target hybrid BDIA/CSR format are abstractly specified. Then the tensor is converted from the original format to the target format. Finally, the SpMV compute is defined with the format-converted tensor as an argument.

IV. Evaluation

We demonstrate the benefits of UniSparse by evaluating the performance of supported custom formats over traditional formats on an Intel CPU, an NVIDIA GPU, and a simulated PIM

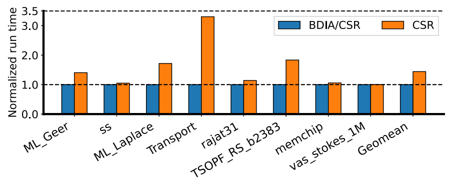


Fig. 6. Normalized run time of SpMV using the BDIA/CSR decomposed by UniSparse vs. Intel MKL using the CSR on a 48-core CPU.

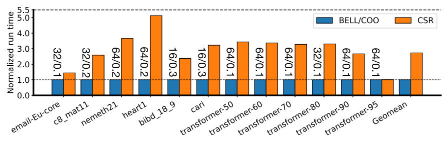


Fig. 7. Normalized run time of cuSPARSE SpMM using the BELL/COO decomposed by UniSparse vs. CSR on NVIDIA A6000. Datasets transformer-50 to 95 are pruned weight matrices of Transformer [11] with sparsity ranging from 50% to 95%.

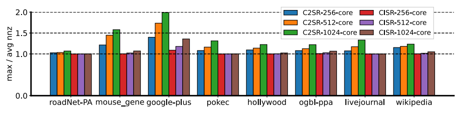


Fig. 8. Load imbalance of SpMV using CISR and C^2 SR formats on different number of PIM cores.

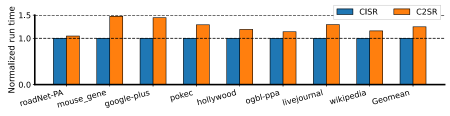


Fig. 9. Normalized run time of SpMV using CISR and C^2 SR formats on 1024 PIM cores.

device. We use sparse matrices from popular datasets including SuiteSparse [7] and OGB [13].

CPU Execution: We test the SpMV kernel with the hybrid BDIA/CSR format on a 48-core Intel Xeon Gold 6248R CPU at 3.00GHz. The compute kernel is parallelized using OpenMP. The baseline is a highly optimized SpMV implementation in Intel MKL using the CSR format. Fig. 6 shows that UniSparse achieves 1.44× speedup in geomean by generating the hybrid format.

GPU Execution: We evaluate sparse matrix-matrix multiplication (SpMM) using the hybrid BELL/COO format and compare it with the one using the CSR format. The compute kernel is deployed on an NVIDIA RTX A6000 GPU through APIs provided by the cuSPARSE library. Fig. 7 shows the comparison of the normalized runtime. The decomposition parameters (block size/density threshold) are shown above each bar of the BELL/COO format. The hybrid BELL/COO format on the selected sparse matrices leads to a 2.7× geomean speedup.

PIM Simulation: We evaluate SpMV using the C^2SR and CISR formats on a simulated PIM device [8] with 256, 512, and 1024 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. Fig. 8 shows the maximum versus the average number of non-zeros. As the number of cores increases, the load imbalance introduced by the C^2SR format gradually becomes a bottleneck, whereas the CISR format mitigates this issue. Fig. 9 shows the normalized runtime of SpMV on 1024 PIM cores using the C^2SR versus the CISR format. Compared with the C^2SR format, using the CISR format improves the performance by 1.24× in geomean.

V. Conclusion

We present UniSparse, an intermediate language for general sparse format customization, and a compiler automating both format pre-processing and compute kernel generation. UniSparse achieves performance improvement over the state-of-the-art with important sparse kernels and custom formats on platforms including CPUs, GPUs and a simulated PIM device.

References

[1] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–11.
[2] A. Bik et al., "Compiler support for sparse tensor computations in MLIR," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–25, 2022.
[3] A. Bik and H. Wijshoff, "Compilation techniques for sparse matrix computations," in *Proc. Int. Symp. Supercomput.*, 1993, pp. 416–424.
[4] J. W. Choi et al., "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 115–126, 2010.
[5] S. Chou et al., "Format abstraction for sparse tensor algebra compilers," in *Proc. Int. Conf. Object-Oriented Program., Syst. Lang. Appl.*, 2018, pp. 1–30.
[6] S. Chou et al., "Automatic generation of efficient sparse tensor format conversion routines," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 823–838.
[7] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.
[8] A. Devic et al., "To PIM or not for emerging general purpose processing in DDR memory systems," in *Proc. Int. Symp. Comput. Archit.*, 2022, pp. 231–244.
[9] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Proc. IEEE Symp. Field Prog. Custom Comput. Mach.*, 2014, pp. 36–43.
[10] T. Fukaya et al., "Accelerating the SpMV kernel on standard cpus by exploiting the partially diagonal structures," 2021, *arXiv:2105.04937*.
[11] T. Gale et al., "The state of sparsity in deep neural networks," 2019, *arXiv:1902.09574*.
[12] D. Guo et al., "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 103–120, 2016.
[13] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," 2020, *arXiv:2005.00687*.
[14] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2021, pp. 1–9.
[15] F. Kjolstad et al., "The tensor algebra compiler," in *Proc. Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2017, pp. 1–29.
[16] K. Kourtis, G. Goumas, and N. Koziris, "Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression," in *Proc. 37th Int. Conf. Parallel Process.*, 2008, pp. 511–519.
[17] C. Lattner et al., "MLIR: A compiler infrastructure for the end of Moore's law," 2020, *arXiv:2002.11054*.
[18] W. Pugh and T. Shpeisman, "SIPR: A new framework for generating efficient code for sparse matrix computations," in *Proc. 11th Int. Workshop Lang. Compilers Parallel Comput.*, 1999, pp. 213–229.
[19] K. Remington et al., "NIST sparse BLAS: User's guide," Citeseer, Tech. Rep., 6744, 1996.
[20] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchit.*, 2020, pp. 766–780.
[21] R. Tian et al., "A high-performance sparse tensor algebra compiler in multi-level IR," 2021, *arXiv:2102.05187*.
[22] Z. Ye et al., "SparseTIR: Composable abstractions for sparse compilation in deep learning," 2022, *arXiv:2207.04606*.