An Intermediate Language for General Sparse Format Customization

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

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. More concretely, we express a sparse format as a map from dense coordinates to a layout tree using a small set of well-defined query and mutation 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 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, specialized application languages, heterogeneous (hybrid) systems, sparse linear algebra

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%. 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/or memory footprint. These sparse formats only store the non-zero elements (or non-zero 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 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 [1], [4], [12] and banked formats for FPGAs [9], [14] and dedicated accelerators [21]. 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 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 [5], [15], COMET [22], and SparseTIR [23] 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. Moreover, attribute-based encoding increases barriers to automating the conversion (and permutation) among different formats, as the coordinate layout of non-zeros is not directly expressed.

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. This abstraction further enables the compiler to formally reason about the correspondence of layouts between different formats and automate the code generation of format conversion. We implement the compiler on top of a multi-level compiler infrastructure, namely MLIR [18]. Our automation flow demonstrates significant productivity improvement and high performance on multiple hardware backends including CPUs, GPUs, and a simulated PIM device [8].

2 Background and Motivation

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

- *Jie Liu, Zhongyuan Zhao and Zhiru Zhang are with the School of ECE, Cornell University, USA. E-mail: {jl3952, zz546, zhiruz}@cornell.edu.*
- *Zijian Ding is with Peking University, China. E-mail: dzj325@gmail.com.*
- *Benjamin Brock and Hongbo Rong are with Intel, USA. E-mail: {benjamin.brock, hongbo.rong}@intel.com.*

2.1 Custom Sparse Formats

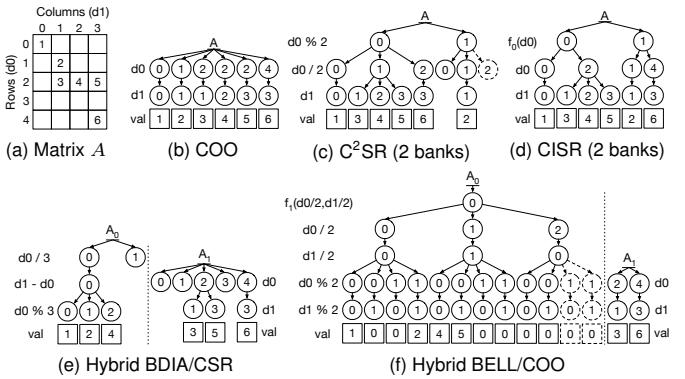


Fig. 1. Selected custom formats of a tensor (here the matrix A). The root of a metadata tree is the symbol of the tensor. A_0 and A_1 are sub-matrices of the matrix A. Circular nodes contain indices inside. Arrows are pointers from a higher major dimension to a lower one. There are index expressions next to a tree, which will be discussed in Section 3.1.

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. 1a. Fig. 1b shows the traditional coordinate (COO) format. Fig. 1c and 1d illustrate two hardware-friendly banking formats: the cyclic channel sparse row (C^2SR) format [21] 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 list-scheduling manner to improve load balancing among compute units accessing these channels. Fig. 1e and 1f 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 the non-zeros along a diagonal for each partition with their offsets in memory. The BELL format stores the non-zeros in blocks and every row has the same number of blocks – it helps improve the performance of sparse compute kernels on GPUs by exploiting data-level parallelism and balancing the work across different rows [4]. By customizing the layout for sub-tensors with different sparsity patterns, hybrid formats can often achieve higher performance than using a single format on the entire tensor [1], [10] .

2.2 Prior Work on Sparse Format Abstraction

Early research on sparse tensor algebra compilers [3], [19] generates compute kernels with hard-coded storage formats. The idea of supporting different data structures with a format abstraction was pioneered by the Bernoulli Compiler [16]. In Table 1, 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 [22] and SparseTIR [23] 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 since attributes do not directly reflect the data layouts.

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. Additionally, this work does not support many specialized formats such as hybrid formats. To the best of our knowledge, UniSparse is the first language-based formulation of format abstraction. Our work complements prior approaches on format customization and decouples format pre-processing from kernel generation.

TABLE 1
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	●	●	●

3 UniSparse: A Unified Abstraction for Customizing Sparse Data Formats

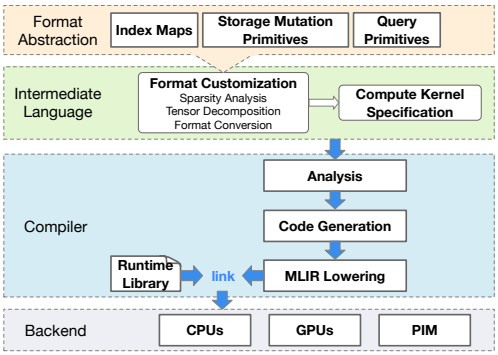


Fig. 2. An overview of the automated compiler flow.

To address the aforementioned limitations of the prior arts, we propose UniSparse, an intermediate language that can formally and concisely describe various sparse formats, facilitating both format conversion and customization (Section 3.1). Fig. 2 outlines the overall compilation flow based on UniSparse. In our approach, a sparse tensor format is encoded as an index map and a set of storage mutation and query primitives, which is decoupled from the compute kernel specification. The compiler decodes how formats are stored internally and generates code for format preprocessing (Section 3.2). Then it can interface with any code generation algorithms to lower compute operations, where we leverage the MLIR SparseTensor work. The intermediate language is implemented as an independent MLIR dialect and lowered to multiple targets, including CPUs, GPUs, and a PIM simulator [8].

3.1 Format Abstraction

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

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. 1a, 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. 1b-1f. For example, the C^2SR format in Fig. 1c is expressed by an index map of `(d0, d1)-> (d0%2, d0/2, d1)`, which indicates that the first dimension of the matrix is divided into two partitions in a cyclic way; the BDIA portion of the hybrid format in Fig. 1e (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. 1d and f_1 for the BELL portion of the hybrid format in Fig. 1f, are such examples. We omit detailed definitions of these indirect maps due to space constraints. At a high-level, function

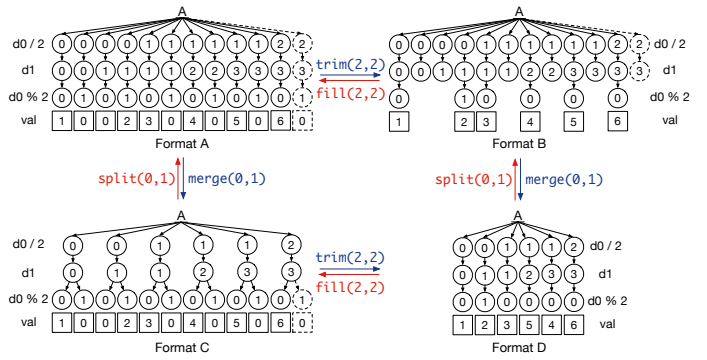


Fig. 3. Examples of storage mutation primitives. Format A, B, C, and D represent four different storage layouts of matrix A in Fig. 1a.

f_0 distributes rows in a way to balance the number of non-zeros among a set of sub-tensors; these sub-tensors will be assigned to different memory channels and processed in parallel. The 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`≤`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.

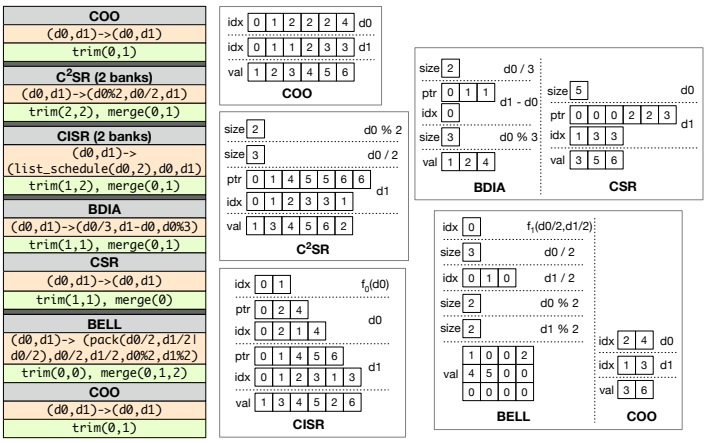
A reduction map divides tensor elements into groups using the same syntax as an index map. However, the destination expression of a reduction map typically has fewer dimensions than the source expression. For example, the reduction map `(d0, d1)-> (d0%2)` assigns the values in the even rows into one group and the values in the odd rows into the other group; another map `(d0, d1)-> (d1-d0)` groups elements on the same diagonals together.

An aggregate function calculates statistics of the groups. We pre-define several aggregation functions: **`count`** returns the number of non-zeros within a group, which can be used to determine sparsity patterns before decomposing a tensor in a hybrid format; **`count-distinct`** computes the number of unique non-zeros within a group, which is useful for tensor value compression [17]; **`min/max`** functions return a minimum/maximum coordinate of a group, useful to express banded formats such as the skyline format [20].

With the above primitives, UniSparse can express both input-aware formats (e.g., hybrid formats in Fig. 1e/1f) and hardware-aware formats (e.g., blocking formats in Fig. 1c/1d). Fig. 4a lists the abstractions of the formats in Fig. 1. We further define format pre-processing operations, **`decompose`** and **`convert`**, to enable declarative specification of tensor decomposition and format conversion. We omit the details here due to space constraints.

3.2 Automation

Analysis. The compiler infers how sparse tensors are stored in memory, i.e., data structures shown in Fig. 4b, from format abstractions in Fig. 4a. 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 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. Indirect maps introduce less regular index patterns and always require an explicit `idx` array. The **`trim`**`(S,E)` primitive also calls for `idx` arrays for level `S` to level `E`, since the indices at these levels are not continuous after being trimmed. The **`merge`** primitive adds a `ptr` array to the descendant levels of specified levels.



(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. Here **`list_schedule`** and **`pack`** are two indirect map functions. **`list_schedule`**`(d0,2)` schedules the rows (`d0`) of a tensor to 2 banks, and **`pack`**`(d0/2,d1/2|d0/2)` packs non-zeros in the same dimension (with the same `d0/2`) together in storage. `|d0/2` marks that the output index is independent of the level `d0/2` and thus, the level `d0/2` is stored in `size`.

Code Generation. The compiler lowers the declarative **`convert`** operation into a sequence of atomic 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. As for indirect maps, we only support converting from regular formats to formats with user-defined map functions, but not vice versa.

3.3 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.

```
1  // Format abstraction
2  #COO = #encoding<{
3    indMap=#map<(i,j)->(i,j)>,stgPrim=#prim<trim(0,1)> }>
4  #CSR = #encoding<{
5    indMap=#map<(i,j)->(i,j)>,stgPrim=#prim<merge(0), trim(1,1)> }>
6  #BDIA = #encoding<{
7    indMap=#map<(i,j)->(i div 50, j-i, i mod 50)>,
8    stgPrim=#prim<merge(0), trim(1,1)> }>
9  #COO_COO = #hybrid<{ fmats=[#COO, #COO] }>
10 #BDIA_CSR = #hybrid<{ fmats=[#BDIA, #CSR] }>
11
12 // Format pre-processing
13 %A1 = unisparse.decompose (%in_A, %thld) {
14   rmap=(i,j)->(i div 50, j-i)}: tensor<?x?xf32, #COO_COO>
15 %A2 = unisparse.convert (%A1): tensor<?x?xf32, #BDIA_CSR>
16
17 // Compute operation
18 #spmv = { indexing_maps = [
19   affine_map<(i,j)->(i,j)>, // for argument %A2
20   affine_map<(i,j)->(j)>, // for argument %in_X
21   affine_map<(i,j)->(i)>], // for argument %out_Y
22   iterator_types = ["parallel", "reduction"] }
23 %0 = linalg.generic #spmv
24   ins(%A2, %in_X : tensor<?x?xf32, #BDIA_CSR>, tensor<?xf32>)
25   outs(%out_Y: tensor<?xf32>) {
26     ^bb0(%a: f32, %b: f32, %x: f32):
27       %2 = arith.mulf %a, %b : f32
28       %3 = arith.addf %x, %2 : f32
29       linalg.yield %3 : f32
30   } -> tensor<?xf32>
```

Fig. 5. A UniSparse program for SpMV. Assume the format of the input tensor A is COO. A **`decompose`** operation in Line 13 divides the tensor into two sub-tensors adaptively by embedding a sparsity query with a reduction map (rmap) in Line 14, and a **`convert`** operation in Line 15 translates the original format into a hybrid BDIA/CSR format. All these formats have been abstractly specified beforehand in index maps (indMap) and storage mutation primitives (stgPrim) in Line 2 - 10. The SpMV kernel is specified using a **`linalg.generic`** operation in Line 23 - 30.

4 Evaluation

We demonstrate the benefits of UniSparse by evaluating the productivity improvement and the performance of supported custom formats over traditional formats on an Intel CPU, an NVIDIA GPU, and a simulated PIM device. We use sparse matrices from popular datasets including SuiteSparse [7] and OGB [13]. We further obtain a set of sparse weight tensors/matrices from a pruned Transformer model [11].

Productivity. A single **`decompose`** operation in the UniSparse program of Fig. 5 lowers to 158 lines of code (LOC) in C++, and an equivalent implementation in Python is 105 LOC. Thus much less effort is required in designing and pre-processing formats; switching between formats is easier too, since the specification of the compute kernel is decoupled from format customization.

CPU Execution. We test the SpMV kernel with the hybrid BDIA/CSR format (program in Fig. 5) on a 48-core Intel Xeon Gold 6248R CPU at 3.00GHz. The compute kernel is parallelized using OpenMP and implemented as a runtime library linked to the program. The baseline is a highly optimized SpMV implementation in Intel MKL using the CSR format. Fig 6 shows the execution time of the kernel normalized against the baseline. Using the hybrid format leads to a 1.44× speedup in geomean.

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 normalized runtime of the SpMM kernel using the CSR format vs. the BELL/COO format. 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. Furthermore, we expect to see a higher speedup through additional parameter tuning in matrix decomposition.

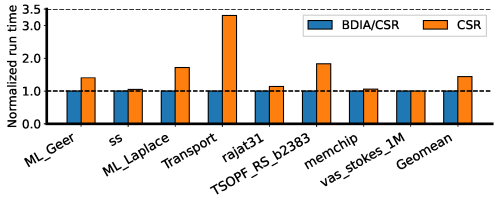


Fig. 6. Normalized run time of SpMV using the BDIA/CSR decomposed by UniSparse vs. Intel MKL using the CSR on a 48-core CPU.

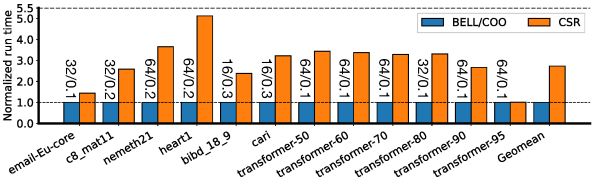


Fig. 7. Normalized run time of cuSPARSE SpMM using the BELL/COO decomposed by UniSparse vs. CSR on NVIDIA A6000. Datasets transformer-50 to 95 are pruned weight matrices of Transformer [11] with sparsity ranging from 50% to 95%.

PIM Simulation. We evaluate SpMV using the C^{2}SR 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 vs. the average number of non-zeros. 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. Fig. 9 shows the normalized runtime 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 1.24× in geomean.

5 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 significant productivity and performance improvement over the state-of-the-art with important sparse kernels and various custom formats on multiple platforms, including CPUs, GPUs and a simulated PIM device.

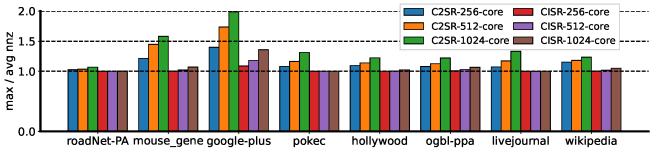


Fig. 8. Load imbalance of SpMV using CISR and C^{2}SR formats on different number of PIM cores.

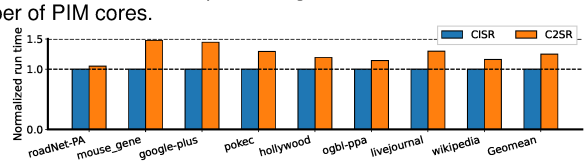


Fig. 9. Normalized run time of SpMV using CISR and C^{2}SR formats on 1024 PIM cores.

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 #2118709 and #2212371, and by AFRL and DARPA under agreement FA8650-18-2-7863.

References

[1] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," *Int'l Conf. on High Performance Computing Networking, Storage and Analysis*, 2009.

[2] A. Bik, P. Koanantakool *et al.*, "Compiler support for sparse tensor computations in MLIR," *ACM Trans. on Architecture and Code Optimization (TACO)*, vol. 19, no. 4, pp. 1–25, 2022.

[3] A. Bik and H. Wijshoff, "Compilation techniques for sparse matrix computations," *Int'l Symp. on Supercomputing (ICS)*, 1993.

[4] J. W. Choi, A. Singh *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, F. Kjolstad *et al.*, "Format abstraction for sparse tensor algebra compilers," *Intl'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.

[6] S. Chou, F. Kjolstad *et al.*, "Automatic generation of efficient sparse tensor format conversion routines," *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[7] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[8] A. Devic, S. B. Rai *et al.*, "To PIM or not for emerging general purpose processing in DDR memory systems," *Int'l Symp. on Computer Architecture (ISCA)*, 2022.

[9] J. Fowers, K. Ovtcharov *et al.*, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[10] T. Fukaya, K. Ishida *et al.*, "Accelerating the SpMV kernel on standard cpus by exploiting the partially diagonal structures," *arXiv preprint arXiv:2105.04937*, 2021.

[11] T. Gale, E. Elsen *et al.*, "The state of sparsity in deep neural networks," *arXiv preprint arXiv:1902.09574*, 2019.

[12] D. Guo, W. Gropp *et al.*, "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 103–120, 2016.

[13] W. Hu, M. Fey *et al.*, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[14] Y. Hu, Y. Du *et al.*, "GraphLily: Accelerating graph linear algebra on hbm-equipped fpgas," *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2021.

[15] F. Kjolstad, S. Kamil *et al.*, "The tensor algebra compiler," *Intl'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.

[16] V. Kotlyar, K. Pingali *et al.*, "Compiling parallel sparse code for user-defined data structures," Cornell University, Tech. Rep., 1997.

[17] K. Kourtis, G. Goumas *et al.*, "Improving the performance of multi-threaded sparse matrix-vector multiplication using index and value compression," *Int'l Conf. on Parallel Processing (ICPP)*, 2008.

[18] C. Lattner, M. Amini *et al.*, "MLIR: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[19] W. Pugh and T. Shpeisman, "SIPR: A new framework for generating efficient code for sparse matrix computations," *Languages and Compilers for Parallel Computing*, pp. 213–229, 1999.

[20] K. Remington, R. Pozo *et al.*, "NIST Sparse BLAS: User's guide," Citeseer, Tech. Rep., 1996.

[21] N. Srivastava, H. Jin *et al.*, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," *Int'l Symp. on Microarchitecture (MICRO)*, 2020.

[22] R. Tian, L. Guo *et al.*, "A high-performance sparse tensor algebra compiler in multi-level IR," *arXiv preprint arXiv:2102.05187*, 2021.

[23] Z. Ye, R. Lai *et al.*, "SparseTIR: Composable abstractions for sparse compilation in deep learning," *arXiv preprint arXiv:2207.04606*, 2022.