2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) | 978-1-6654-1134-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/LLVMHPC54804.2021.00009

A High Performance Sparse Tensor Algebra Compiler in MLIR

Ruiqin Tian*, Luanzheng Guo*, Jiajia Li †, Bin Ren†, Gokcen Kestor*‡
*Pacific Northwest National Laboratory, †William & Mary, ‡University of California, Merced
*{ruiqin.tian, lenny.guo, gokcen.kestor}@pnnl.gov, †{jli49, bren}@wm.edu, ‡ gkestorgioiosa@ucmerced.edu

***Abstract*—Sparse tensor algebra is widely used in many applications, including scientific computing, machine learning, and data analytics. The performance of sparse tensor algebra kernels strongly depends on the intrinsic characteristics of the input tensors, hence many storage formats are designed for tensors to achieve optimal performance for particular applications/architectures, which makes it challenging to implement and optimize every tensor operation of interest on a given architecture. We propose a tensor algebra domain-specific language (DSL) and compiler framework to automatically generate kernels for mixed sparse-dense tensor algebra operations. The proposed DSL provides high-level programming abstractions that resemble the familiar Einstein notation to represent tensor algebra operations. The compiler introduces a new Sparse Tensor Algebra dialect built on top of LLVM's extensible MLIR compiler infrastructure for efficient code generation while covering a wide range of tensor storage formats. Our compiler also leverages input-dependent code optimization to enhance data locality for better performance. Our results show that the performance of automatically generated kernels outperforms the state-of-the-art sparse tensor algebra compiler, with up to 20.92x, 6.39x, and 13.9x performance improvement over state-of-the-art tensor algebra compilers, for parallel SpMV, SpMM, and TTM, respectively.**

I. Introduction

Tensor algebra is at the core of numerous applications in scientific computing, machine learning, and data analytics. Tensors are a generalization of matrices to any number of dimensions, which are often large and sparse. Sparse tensors are used to represent a multi-factor or multi-relational dataset, and have found numerous applications in data analysis and mining [1], [2] health care [3], natural language processing [4], machine learning [5], and social network analytics [6], among many others.

Developing optimized kernels for sparse tensor algebra methods is challenging [7], [8], [9]. First, sparse tensors are often stored in a compressed form (indexed data structures) and computational kernels need to efficiently loop over the nonzero elements of the tensor inputs. Second, iterating over nonzero elements highly depends on the particular storage format employed, hence many algorithms exist to implement the same operation, each targeting a specific format. Finally, applications may use multiple formats concurrently throughout the computation and mix different formats in the same operation to achieve high performance [10].

Current solutions implement ad-hoc approaches for particular computer architecture and/or format. Most of these algorithms tackle specific problems and domains and conveniently store sparse tensors in a format that exploits the characteristics of the problem. This approach has originated tens of different formats [11], [12], [13], [10] to represent sparse tensors. This makes it infeasible to manually write optimized code for each tensor expression considering all possible combinatorial combinations of tensor operations, formats, and architectures.

To solve the above challenges, researchers have developed early compilers to support automatic code generation for sparse kernels starting from the loop representation [14], [15]. TACO [16], [17], [7] is a state-of-the-art source-to-source compiler that automatically generates sparse tensor algebra kernels starting from high-level index notations. The TACO compiler efficiently supports any compound expression with many different storage formats. Despite its broad success, TACO commits to a limited set of source (C++, Python) and destination (C++, OpenMP, and CUDA) languages. Each code generation path is independent, thus many optimizations cannot be shared. Moreover, TACO directly translates source C++ code to OpenMP or CUDA kernels, employing only a reduced set of domain-specific optimizations. Although transforming the code into independent parallel programming languages that natively execute on selected architectures results in good performance, the overall approach shows limited reusability of software components, reduced interoperability and composability, and limited support for additional architectures.

We present a sparse tensor algebra compiler implemented on top of our prior work, COMET [18]. Our compiler includes a highly-productive language that provides high-level programming abstractions that resemble the Einstein notations to represent tensor operations while supporting various tensor storage format. The compiler automatically generates efficient code for a given mixed dense/sparse tensor expression by leveraging high-level abstractions. The multi-level intermediate representation employed in this work brings several advantages: 1) allowing for the co-existence of multiple abstractions within the same compilation framework with interoperable representations, 2) breaking the isolation between domains and enabling comprehensive optimizations, 3) enabling high-level representation to progressively lowered and optimized to lower-level abstractions to target various architectures.

The compilation process starts from the high-level sparse and dense tensor algebra (TA) DSL. To enable modular code generation with respect to formats and combinations of formats, we introduce an internal sparse storage format in the Multi-Level Intermediate Representation (MLIR) infrastructure [19] based on a limited set of attributes associated with each dimension [16]. By properly combining those attributes in each dimension, users can easily express common sparse tensor compressed formats, such as COO [20], CSR [21], DCSR [22], ELLPACK [23], CSF [11] and Mode-generic [10]. This approach lets users not only mix and match storage format desired for their applications but also enables custom formats without modifying the underlying compiler infrastructure. The proposed Sparse Tensor Algebra dialect is the first dialect in the compiler stack, which includes tensor algebra specific data types, storage format, and operations. At this level, the proposed compiler performs an input-dependent optimization to take advantage of input sparse data patterns. In particular, a state-of-the-art data reordering algorithm [24] is employed to increase spatial and temporal locality for performance improvement. The sparse TA dialect is then lowered to Structured Control Flow (SCF) dialect, loop form, and arithmetic operations (Standard) dialect, all of which are readily available in MLIR [19]. During this lowering, the compiler analyzes the dimension attributes represented in the TA dialect and produces code to efficiently iterate over the nonzero elements of the input tensors for each dimension. An iteration graph [7] is used to describe how to effectively iterate over the non-zero values of a tensor expression. Once the loop form of a computation has been generated at the Intermediate Representation (IR) stack, our compiler either lowers the code for sequential or parallel execution. In the former case, it produces a high-quality LLVM IR (which enables better loop unrolling and vectorization (Section VII)); in the latter case, instead, it lowers code to the async dialect for asynchronous task execution based on LLVM co-routines. Compared to hand-tuned libraries [8], [12], [11] and source-to-source compilers [7], [17], [25], our approach is more portable, flexible, and adaptable, as emerging architectures and storage formats can be added without re-engineering the computational algorithms.

To the best of our knowledge, this work introduces the *first* MLIR-based sparse tensor algebra compiler that introduces generic code generation for arbitrary input formats, data reordering, and automatic parallelization within the same framework. The proposed compiler can improve end-user application performance while supporting efficient code generation for a wider range of formats specialized for different application and data characteristics. This paper makes the following contributions:

- An approach to designing a modular domain-specific compiler in multi-level IRs for sparse tensor algebra.
- A tensor algebra language and its representative sparse dialect in MLIR, which encodes the high-level abstraction required for sparse kernels.
- The integration of internal sparse storage format based on the format attribute per dimension within in MLIR to support a wide range of sparse storage formats.
- An efficient code generation for sparse tensor expressions while employing input-dependent code optimization to enhance data locality.
- An comprehensive performance evaluation of the automatically generated code with the proposed compiler. The comparison, based on 2833 input matrices and six tensors, shows that this work significantly outperforms the state-of-the-art TACO compiler.

II. Background and Motivation

There exist various compressed and uncompressed formats to store sparse matrices and tensors, including COOrdinate (COO), Compressed Sparse Row (CSR), Double Compressed Sparse Row (DCSR), ELLPACK, Compressed Sparse Fiber (CSF), and Mode-Generic. The specific format chosen to represent data in an application generally depends on the expected characteristics of the data itself and how these impact other desired properties, such as performance of a computational kernel or memory footprint.

Each format is important for different reasons. COO [20] is commonly used to store sparse matrices and tensors, such as the Matrix Market exchange format [26] and the FROSTT sparse tensor format [27]. While COO is the most natural format, it is not necessarily the most performant format. CSR [21] is for sparse matrices, which compresses row indices as pointers to row beginning positions to avoid duplicated storage and increase performance for memory bandwidth-bound computation such as Sparse-Matrix Dense-Vector (SpMV). DCSR [22] further compresses zero rows by adding an extra pointer to nonzero rows based on the CSR format. With an extra level of compression on rows, DCSR is more efficient than CSR for highly sparse (hypersparse) data. The ELLPACK [23] format is efficient for matrices that contain a bounded number of nonzeros per row, such as matrices that represent well-formed meshes. CSF [11] generalizes the DCSR or CSR matrix format to high-order tensors that compresses every dimension. Mode-Generic format [10] is a generic representation of semi-sparse tensors with one or more dense dimensions stored as dense blocks with the coordinates of the blocks stored in COO. An application might need any or even several of these formats based on its needs, which makes it important to support computation with various tensor storage formats and their combinatorial combinations. The main challenge for a portable compiler is how to generate efficient code to compute any tensor algebra expression for any combination of distinct formats. This problem is especially complicated for expressions that involve multiple operands.

Because of the large number of storage formats and possible combinations, most state-of-the-art sparse tensor libraries [28], [29] support only a few sparse formats (and generally only binary operations) or convert tensors to an internal storage format, thereby potentially losing the performance, memory footprint, or other advantages that a specific format may offer. Moreover, libraries often support a limited set of architectures because of the need of resorting to assembly intrinsics to achieve high performance. Compilers, on the other hand, can automatically generate the efficient code for specific input formats and their combinations, increasing flexibility, adaptivity to new formats, and portability to various hardware platforms.

```
 1 def main() {
 2   #IndexLabel Definition
 3   IndexLabel [a] = [?];
 4   IndexLabel [b] = [?];  #IndexLabel [b] = [0:?];
 5   IndexLabel [c] = [32]; #IndexLabel [c] = [0:32:1];
 6
 7   #Tensor Definition
 8   Tensor<double> A([a,b],CSR);   #Tensor<double> A([a,b],{D,CU});
 9   Tensor<double> B([b,c],Dense);#Tensor<double> B([b,c],{D,D});
10   Tensor<double> C([a,c],Dense);#Tensor<double> C([a,c],{D,D});
11
12   #Tensor Readfile Operation
13   A[a,b] = rt_read(filename);
14
15   #Tensor Fill Operation
16   B[b,c] = 1.0;
17   C[a,c] = 0.0;
18
19   #Tensor Contraction
20   C[a, c] = A[a,b] * B[b,c];
21 }
```
Listing 1: An example program in the TA language for Sparse Matrix-times-Dense-Matrix operation.

To achieve this goal, two important requirements need to be satisfied: 1) a unified way to represent important sparse storage formats (Section IV) and 2) an efficient algorithm to generate specific code for a given expression and its particular input formats (Section VI).

III. Overview

This paper introduces a DSL and compiler that capture and explicitly preserve high-level information throughout the compilation process to generate efficient code for modern architecture. The compiler performs a progressive lowering process to map high-level operations to low-level architectural resources, a series of optimizations performed in the lowering process, and various IR dialects to represent key concepts, operations, and types at each level of the multi-level IR. This section reviews the key characteristics of our compiler framework. The proposed compiler is built on top of our previous work, COMET [18], by leveraging the MLIR framework [19]. MLIR is a compiler infrastructure to build reusable and extensible compilers and IRs. MLIR supports the compilation of high-level abstractions and domain-specific constructs and provides a disciplined, extensible compiler pipeline with gradual and partial lowering. Users can build domain-specific compilers and customized IRs (called dialect), as well as combining existing IRs, opting into optimizations and analysis.

The code generation in this work follows a progressive lowering approach where optimizations are applied at different levels. Figure 1 shows the proposed compilation pipeline, where our contributions in the MLIR infrastructure are annotated by the dashed box. Programmers express their computation in a high-level tensor algebra Domain-Specific Language (DSL) (Section V), such as the SpMM program shown in Listing 1. In the Listing, three tensors, `A`, `B`, and `C` are defined (Lines 8-10). Tensor `A` is a sparse tensor stored in CSR format while the other two are dense tensors. The DSL allows users to express tensor's properties for each dimension. `IndexLabels` represent generic dimensions (lines 3-5) and

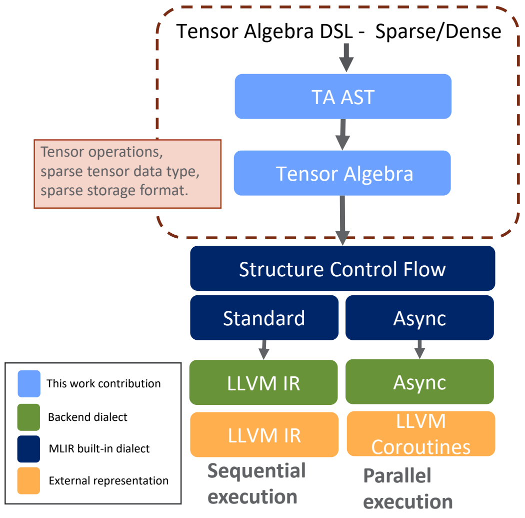


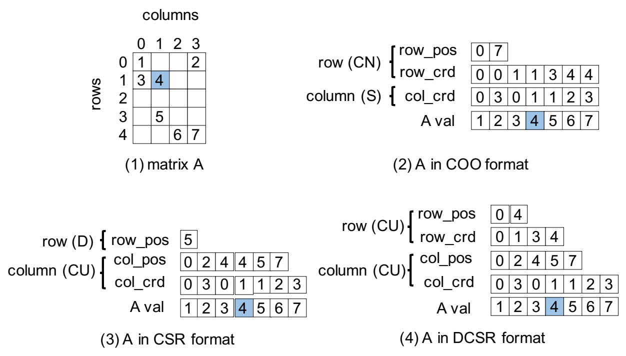
Fig. 1: The proposed execution flow and compilation pipeline

are associated with a cardinality. For example, in Listing 1, index label `a` and `b` have an undetermined cardinality (i.e., the index labels are dynamic) while index label `c` has cardinaltiy 32 (i.e, static index label). When a tensor is defined, sparsity properties are associated to each dimension to represent the sparse storage format. For example, for tensor `A`, the first dimension is dense (`D`) and the second dimension is sparse (`CU`). The DSL provides convenient macros to express this format as CSR in the program.

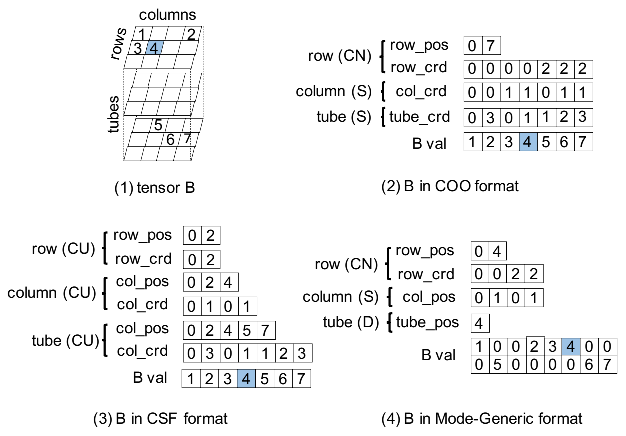
During the progressive lowering, the language operators, types, and structures are first mapped to an abstract syntax tree and then lowered to the proposed sparse Tensor Algebra (TA) dialect. The TA dialect contains domain-specific concepts, such as multi-dimensional sparse/dense tensors, data type, sparsity properties, and tensor expressions. Our compiler framework applies high-level optimizations and transformations leveraging semantics information carried from the DSL such as data reordering to enhance data locality. Next, the compiler lowers the sparse TA dialect to lower levels of dialects in the compilation pipeline by considering the format attribute per dimension and the actual tensor expression (Section VI-B). Tensor algebra operations (line 20 in Listing 1) are lowered to SCF dialect which represents loop form and the Standard dialect for arithmetic operations in the core MLIR framework. At this point, the compiler employs generic optimizations during the lowering steps but also considers additional information about the final target architecture. For CPU execution, the code is lowered directly to the Low-Level Virtual Machine (LLVM) dialect for sequential execution or to the async dialect for asynchronous task parallel execution first and then to LLVM IR for final assembly and linking.

IV. Tensor Storage Format

Our objective is to develop a unified and generic algorithm to generate efficient code independently of the input tensors'



(a) Sparse Matrix



(b) Sparse Tensor

Fig. 2: Example matrix and tensor represented in different formats. Each format is a combination of the storage format attributes.

storage format. To this extend, our compiler needs to store tensors in a uniform way. This internal storage formats need to preserve the characteristics of the original format, e.g., data compression or performance for specific sparse patterns, while allowing a unified algorithm to generate efficient code for each computational expression. Representing the sparsity property of tensor per dimension separately has been shown to be an effective way to *generalize* tensor storage formats [30], [31], [11]. Representing each dimension independently makes it easier to manage, adapt, and represent various storage formats and to generate computational kernels uniformly.

Our work borrows this approach and defines a set of storage format attributes for each dimension as a sparsity property of a tensor in the TA dialect. Code generation is then based on each dimension's storage format attributes rather than the whole format, which greatly reduces the number of formats and combinations that a compiler needs to support. Importantly, we do not convert the original data layout into a different storage format. Instead, the storage format attributes are used to compose meta-data information that describes the original format, i.e., the data layout of the original format is preserved in memory and retains the original characteristics (compression, locality, etc.).

Figure 2 shows examples of how the internal storage format constructed in the compiler for sparse matrices and tensors, respectively, with the representation of varied storage format attributes combinations. The following is the definition of four storage format attributes used in this work:

Dense (D). This dimension is in the "dense" format, i.e., all elements (zeros and non-zeros) are stored and can be directly accessed during the computations.

Compressed_Unique (CU). This dimension is in a "compressed unique" format, i.e., the coordinates of nonzero elements in this dimension are compressed and only the unique ones (no duplication) are stored.

Compressed_Nonunique (CN). This dimension is in a "compressed non-unique" format, i.e., all the coordinates of nonzero elements will be recorded.

Singleton (S). The dimension is in a "singleton" format, i.e., all the nonzero coordinates are recorded.

Internally, each tensor dimension is described by two arrays, a position (`pos`) and a coordinate (`crd`) array. **D** only uses the `pos` array to store the size of the dimension as a scalar value, such as the `row` dimension in Figure 2a(3). The compressed storage format attributes **CU** and **CN** use both `pos` and `crd` arrays to store the nonzero coordinates and their positions. **CU** format stores unique values in the `crd` array and uses `pos` to store the start position of each unique coordinate, such as the `row` dimension Figure 2a(4), where the elements 1 and 2 are in the same row, but only one row coordinate is stored in `row_crd` array. **CN** stores all the coordinates of the nonzero elements in `crd` array, which will be accessed one by one. CN then stores the start and the length of the `crd` array to the `pos` array, such as the `row` dimension Figure 2a(2), where all the row coordinates of the nonzeros are stored in `row_crd` array, `row_pos` only stores the start and the length of the `row_crd` array. **S** only uses the `crd` array to store the nonzero coordinates in the dimension. We can represent the important sparse storage formats in a uniform way by properly combining the tensor storage format attributes, including COO, CSR, DCSR, BCSR, CSB, ELLPACK, CSF and Mode-generic while retaining each format's characteristics.

V. High-level Language Definition

This work introduces a high-level Tensor Algebra DSL that increases portability and productivity by allowing scientists to reason about the implementation of their algorithms in their familiar notations and syntax. Specifically, the DSL allows scientists 1) to express concepts and operations in a form that closely resembles their familiar notations and 2) to convey domain-specific information to the compiler for better program optimization. For example, our language represents Einstein mathematical notation and provides users with an interface to express tensor algebra semantics. The same program can be lowered to different architectures, and the lowering steps can follow different optimizations and lowering algorithms, allowing the compiler to produce high-quality code for target architectures without excessive burden on the programmer (see Section VI). We have extended our previous work [18], which is designed for dense tensor computation, to support sparse tensor algebra operations, dynamic index labels, sparse storage format attributes, and runtime utility functions.

As discussed above, Listing 1 shows an example of a program. In our DSL, a tensor object refers to a multi-dimensional array of arithmetic values that can be accessed by indices. Range-based index label constructs (`IndexLabel`) represent the range of indices expressed through a scalar, a range, or a range with increment. Index labels can be used both for constructing a tensor or for representing a tensor operation. Different from the previous work, `IndexLabels` can be defined as *static* or *dynamic* to support index variables whose sizes are unknown at compile time. Static `IndexLabels` explicitly state the size of the dimension (Line 5 in Listing 1) while dynamic `IndexLabels` (Lines 3 and 4) only indicate that there exists a dimension, but the size will be determined later on during the execution of the program. Dynamic and static index labels differ in that dynamic index labels indicate an unknown size through a question mark (`?`) operator while static index labels explicitly state the size of the dimension through a scalar value.

A tensor is constructed by defined static or dynamic index labels and by declaring the sparsity property of each dimension, according to the internal storage format described in the previous section. In Listing 1 tensor `A` is stored in CSR format, while tensors `B` and `C` are stored in dense format. Note that our work provides convenient notation to represent the most common tensor storage format, avoiding the need to specify the storage format for each dimension, as described in the comments at Lines 8-10. Internally, however, the compiler reasons in terms of sparsity property associated for each dimension when generating code.

In the program in Listing 1, the tensor `A`, `B`, and `C` are initialized with a tensor file by `rt_read()`, the constant value `1.0`, and the constant value `0.0`, respectively. The function `rt_read()` first reads a tensor from the file in Matrix Market format [32] and then converts it to our internal storage format (see Section IV) to represent `CSR`. We implement `rt_read()` as a *runtime function*, which can be called in the program directly to read matrices/tensors from files.

The last line in the program performs the SpMM operation. Programmers need not explicitly state that the operation is an SpMM such as when invoking a library call. They can simply use the common tensor contraction `*` operator for the tensor expression. The compiler will infer that the operator refers to an SpMM operation from the index notation, in this case, a sparse matrix and a dense matrix, and will generate the proper code to iterate over the specific storage format through rules generated from the definition of storage format attributes. Also, note that we employs index labels to determine the type of operation to perform. For example, the `*` operator refers to a tensor contraction if the contraction indices are adjacent or to element-wise operation otherwise. In Listing 1, the index label `b` is used as contraction indices between `A` and `B` (adjacent or internal indices), thus the operator `*` refers to a tensor contraction. Conclusively, the our TA language simplifies writing tensor algebra programs by supporting common programming paradigms and enables users to express high-level concepts in their familiar notations.

```
1  #map0 = affine_map<(d0, d1, d2) -> (d0, d1)>
2  #map1 = affine_map<(d0, d1, d2) -> (d1, d2)>
3  #map2 = affine_map<(d0, d1, d2) -> (d0, d2)>
4  module {
5    func @main() {
6       %c0 = constant 0 : index
7       %c1 = constant 1 : index
8       %c32 = constant 32 : index
9       %a = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
10      %b = "ta.index_label_dynamic"(%c0, %c1) : (index, index) -> !ta.range
11      %c = "ta.index_label_static"(%c0, %c32, %c1) : (index, index, index) -> !ta.range
12      %A = "ta.tensor_decl"(%a, %b) {format = ["D", "CU"]} : (!ta.range, !ta.range) -> tensor<?x?xf64>
13      %B = "ta.tensor_decl"(%b, %c) {format = ["D", "D"]} : (!ta.range, !ta.range) -> tensor<?x32xf64>
14      %C = "ta.tensor_decl"(%a, %c) {format = ["D", "D"]} : (!ta.range, !ta.range) -> tensor<?x32xf64>
15      "ta.fill_from_file"(%A) {filename="dataset.mtx"}: (tensor<?x?xf64>) -> ()
16      "ta.fill"(%B) {value = 1.0 : f64} : (tensor<?x32xf64>) -> ()
17      "ta.fill"(%C) {value = 0.0 : f64} : (tensor<?x32xf64>) -> ()
18      "ta.tc"(%A, %B, %C) {alpha = 1..000000e+00 : f64, beta = 0..000000e+00 : f64,
               format = [["D", "CU"], ["D", "D"], ["D", "D"]],  indexing_maps = [#map0, #map1, #map2]} :
               (tensor<?x?xf64>, tensor<?x32xf64>, tensor<?x32xf64>) -> ()
19      "ta.return"() : () -> ()
```

Fig. 3: Generated sparse tensor algebra dialect for SpMM.

VI. Compilation Pipeline

This section describes the compiler framework, which consists of two main parts: 1) a sparse TA dialect in MLIR to represent the high-level abstraction required for sparse kernels, and 2) code generation algorithms to generate efficient serial and parallel code starting from the TA DSL.

A. Sparse Tensor Algebra Dialect

We introduce sparse tensor algebra dialect in MLIR, which we call Tensor Algebra (TA) dialect, to support mix dense/sparse tensor algebra computation with a wide range of storage formats. The main goal of this dialect is to represent basic building blocks of the tensor algebra computation, describe tensor algebra specific types and operations, and represent semantics information expressed through the TA DSL. The sparse tensor algebra dialect includes tensor algebra specific data types, attributes, storage format, and operations. Figure 3 shows the generated tensor algebra dialect for the SpMM program in Listing 1. The "ta" prefix in Figure 3 identifies the operations and types from TA dialect. The rest of this section details these operations and types.

Static/Dynamic Index Labels. The sparse tensor algebra dialect supports two types of index label operations, static and dynamic. If the dimension size of the index is known at compile-time, statically defined at the DSL level, the compiler generates `ta.index_label_static` operation (Line 11 in Figure 3). This operation has three operands, which represent the start, end, and step value on this index (see comment in Listing 1). `ta.index_label_dynamic` operations (Line 9) are used to represent index labels for

```
%A = “ta.sptensor_construct”(%A1pos, %A1crd, %A2pos, %A2crd, %Aval) :
        (tensor<?xi32>, tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
        tensor<?xf64>)  ->
        (!ta.sptensor<tensor<?xi32>, tensor<?xi32>, tensor<?xi32>,
        tensor<?xi32>, tensor<?xf64>>)
```

Fig. 4: Sparse tensor data structure construction operation

those dimensions that are unknown at compile time. Different from static index labels, a dynamic index label has only two operands, start and step value on this index. The end value will be determined at runtime.

Sparse Tensor Declaration. In the sparse tensor algebra dialect, tensors are declared with `ta.tensor_decl` operations (Line 12), which take an arbitrary number of index labels as operands. Tensor declarations also contain the storage format attributes of the tensor for each dimension. As described above, our language provides utility macros to simplify the description of the sparsity format of each dimension.

Sparse Tensor Operations. The sparse TA dialect also defines the tensor algebra operations. For example, the tensor contraction `ta.tc` operation for an SpMM computation (Line 20 in Figure 3) takes two input tensors and computes the result of the contraction. The first and second operands (%A and %B) are input tensors, and the third operand (%C) is the output tensor. We introduce `formats` attribute to provide the storage format information of each input tensor in the tensor contraction operation. The `indexing_maps` attribute represents the indices of each tensor, which is used to derive the actual index notation of evaluated tensor contraction. The `indexing_maps` also helps propagate indices information down the IR stack. The tensor expression and the storage format information will be further propagated down to the lower level of the IR to provide the format attribute in each dimension when generating the computational code.

Sparse Tensor Data Type. As described in Section IV, a tensor T consists of k dimensions d_i for $0 \leq i \leq k-1$, where every dimension d_i is associated with a uniform storage attribute $a_i \in \{\text{D, CU, CN, S}\}$. We employ two arrays (`crd` and `pos`) for each dimension to describe the storage format. In the TA dialect, we define a sparse tensor as a struct that contains the dimensions' nonzero indices and values.

Figure 4 shows how a 2D sparse matrix is represented in our TA dialect. In Figure 4, `ta.sptensor_construct` is the function to construct the sparse tensor struct, which is implemented as an operation in the TA dialect. The `sptensor_construct` operation takes the `pos` and `crd` arrays in each dimension (%A1pos, %A1crd, %A2pos, %A2crd) and the nonzero values (%AVal) as input, and returns a `ta.sptensor` sparse data type. The tensor types within `ta.sptensor` represent the `pos` and `crd` arrays corresponding to each dimension of the tensor itself (Section IV).

B. Sparse Code Generation Algorithm

The compiler lowers the code from high-level TA language to low-level machine code in multiple lowering steps.

DSL Lowering. The first step in our compilation pipeline consists of lowering the high-level DSL into the sparse TA dialect. Figure 3 shows the TA dialect corresponding to the code presented in Listing 1 in our TA language. The index labels operations in the DSL are lowered to either into a `ta.index_label_static` operation or a `ta.index_label_dynamic` operation (e.g., Lines 9-11 in Figure 3) based on whether the size of the dimension represented by the index label is known or unknown at compile time. The `IndexLabels` at Lines 3-4 of Listing 1 have an unknown size, so they will be lowered into `ta.index_label_dynamic` operations, which only contain the start values of the dimension. The tensor declaration operations in the DSL are lowered to `ta.tensor_decl` operations which contain the format of tensors (e.g., Lines 12-14 in Figure 3). The tensor read file operation in the DSL is lowered to `ta.fill_from_file` operation, which contains the file name information (e.g., Lines 15 in Figure 3). The tensor fill operations, instead, are lowered to `ta.fill`, which include the values to fill the tensors (e.g., Lines 16-17 in Figure 3). The tensor contraction operation is lowered to `ta.tc` operation. The operands of `ta.tc` operation are input and output tensors, the `format` attribute representing the format of each tensor and the `indexing_maps` attribute, which represents the indices contained in each tensor (e.g., Lines 18 in Figure 3).

Progressive Lowering. Next, the sparse TA dialect is further lowered to other MLIR built-in dialects for code generation. We describe this lowering process in two parts, *early* lowering and *late* lowering.

At the early lowering step, the compiler lowers all the operations, except the `ta.tc`, in the sparse TA dialect operation to lower MLIR dialects (e.g., `std` and `linalg`). In particular, for dense tensors, the `ta.tensor_decl` operations are lowered to `alloc` and `tensor_load` operations in the standard dialect. For sparse tensors, `ta.tensor_decl` operations are lowered into a combination of `alloc` and `tensor_load` operations for `pos`, `crd`, and `val` arrays. These coordinates of nonzeros are later used by `ta.sptensor_construct` operation (Figure 4) to construct a sparse tensor. Tensors can be initiliazed in various ways. The tensor values can be read from disk. In this case, the `ta.fill_from_file` operation is invoked to call the runtime `rt_read()` function. Alternatively, the `ta.fill` operation initializes dense tensors using the same scalar to fill all elements of the tensors. The `ta.fill` operation will be lowered into the `fill` operation in the MLIR linalg dialect. Once all dimensions of a sparse tensor are known, the `ta.index_label_dynamic` operations is lowered into the `ta.index_label_static`.

In the late lowering step, only `ta.tc` operations are lowered into the MLIR SCF dialect operations. Figure 5 describes the lowering algorithm for tensor contraction operations with an example mix sparse/dense tensor contraction operation, where a sparse tensor A is multiplied a dense tensor B. The resulting output can be either sparse or dense. The output IR generated is shown in Figure 6. The right-most numbers in

```
# TensorExpr e.g. Cik=Aij*Bjk; Format e.g.A[D, CU], B[D, D], C[D, D]
 CodeGen(TensorExpr, Format)     :
1.   Generate iteration graph iGraph with the tensor expression information
2    Extract format attr for each index, then put them into formats
     # Define variables to store coordinates in the value array of each tensor
3.   Value-Indices $vIdx_A$, $vIdx_B$, $vIdx_C$ = 0

     # Generate for loops based on the format (iterate over all indices in the iteration graph)
4.   for d in iGraph do
5.     switch(formats(d))
6.       case D:  emit-for(arg = 0 to d_pos[0])
7.                  $vIdx_T$ = $vIdx_T$ * d_SIZE + arg          # T ∈ {all tensors that contain index d}
         # m is 0 when d is the first index in the input tensor;
         # Otherwise, m is the argument of the upper-level loop
8.       case CU: emit-for(arg = d_pos[m] to d_pos[m + 1])
9.                  emit-load(d_crd, arg)
10                  $vIdx_T$ = arg       # T ∈ {all sparse tensors that contain index d}
11.                 $vIdx_T$ = $vIdx_T$ * d_SIZE + d_crd[arg]   # T ∈ {all dense tensors that contain index d}
12.      case  CN: emit-for(arg = d_pos[0] to d_pos[1])
13.                 emit-load(d_crd, arg)
14                  $vIdx_T$ = arg      # T ∈ {all sparse tensors that contain index d}
15                  $vIdx_T$ = $vIdx_T$ * d_SIZE + d_crd[arg]  # T ∈ {all dense tensors that contain index d}
16.      case S: arg = argument of upper-level loop
17.               emit-load(d_crd, arg)
18.               $vIdx_T$ += 0          # T ∈ {all sparse tensors that contain index d}
19.               $vIdx_T$ = $vIdx_T$ * d_SIZE + d_crd[arg]     # T ∈ {all dense tensors that contain index d}

     # Generate the loop body of the innermost loop
20   emit operations to do computation for C[$vIdx_C$] += A[$vIdx_A$] *  B[$vIdx_B$]
     # Generate load, mul, store operations
```

Fig. 5: Code generation algorithm

```
1   %A1SIZE_i32 = load %A1pos[%c0] : memref<?xi32>
2   %A1SIZE = index_cast %A1SIZE_i32 : i32 to index                  6-7
3   scf.for %i = %c0 to %A1SIZE step %c1 {
4     %next_i = addi %i, %c1 : index
5     %A2pos_start_i32 = load %A2pos[%i] : memref<?xi32>
6     %A2pos_start = index_cast %A2pos_start_i32 : i32 to index
7     %A2pos_end_i32 = load %A2pos[%next_i] : memref<?xi32>          8-9
8     %A2pos_end = index_cast %A2pos_end_i32 : i32 to index
9     scf.for %arg1 =   %A2pos_start to %A2pos_end step %c1 {
10      %j_i32 = load %A2crd[%arg1] : memref<?xi32>
11      %j = index_cast %j_i32 : i32 to index
12      scf.for %k = %c0 to %c32 step %c1 {                          6-7
13          %Avalue = load %Aval[%arg1] : memref<?xf64>
14          %Bvalue = load %B[%j, %k] : memref<?x32xf64>
15          %product = mulf %Avalue, %Bvalue : f64                   20
16          %Cvalue_old = load %C[%i, %k] : memref<?x32xf64>
17          %Cvalue = addf %Cvalue_old, %product : f64
18          store %Cvalue, %C[%i, %k] : memref<?x32xf64>
19  }}}
```

Fig. 6: Generate `scf` dialect code for SpMM in the CSR format. The right side numbers represent line numbers of the code generation algorithm in Figure 5.

Figure 6 correspond to the lines numbers in the algorithm described in Figure 5.

Our algorithm takes a tensor contraction operation as input, and automatically generates the computational kernel code of a combination of SCF and std dialects. `ta.tc` is the sparse tensor algebra dialect of the tensor contraction operation presented at Line 18 in Figure 3. As shown at Line 18 in Figure 3, `ta.tc` operation is lowered based on the code generation algorithm in Figure 5.

In details, our code generation algorithm consists of three key steps. This algorithm is general, applicable to varied tensor algebra operations, and can generate arbitrary index permutations. Moreover, in contrast to TACO, this work can generate sparse output, which will reduce memory footprint. The basic idea of this code generation is as follows:

Step-I (Line 1 to Line 3) first generates the *iteration graph* with the tensor expression information. The iteration graph is built by considering the access order of each tensor. For example, for tensor A_{ij}, it first access index i, then index j but for tensor A_{ji}, it first access index j, then index i. The generated iteration graph represents the index order to access each tensor. Figure 7 shows some iteration graphs examples, where the circles represent the indices and the rectangle besides each circle represents the tensors that the index belongs to. For example, the iteration graph of tensor expression $C_{ik} = A_{ij} * B_{jk}$ is shown in Figure 7(a), the first accessed index is i in $A1$ (i.e. first dimension of tensor A) and $C1$, the second is j in $A2$ and $B1$, finally k in $B2$ and $C2$.

After building the proper iteration graph for a given tensor expression, the code generation algorithm analyzes the format attribute information for each index in the iteration graph. The format attribute of each index is decided by the usage of this index. If this index appears in dense input tensors only, its format attribute is D; otherwise, the format attribute is decided by the corresponding dimension of the sparse tensor. For the above tensor expression example, the format attribute of index i is D and j is CU (both decided by sparse input tensor A), and k is D (decided by dense input tensor B), respectively. After collecting this information, we define three index variables ($vIdx_A$, $vIdx_B$ and $vIdx_C$) to access the value array of tensor A, B and C, respectively (Line 3).

Step-II (Line 4 to Line 19) iterates over each dimension to generate loop structure code (as the algorithm line starting with "emit" shows). It leverages the aforementioned definition of each storage format attribute to find nonzero coordinates in each dimension via `pos` and `crd` arrays (i.e., d_pos and d_crd in the algorithm). Next, the algorithm gets the coordinates from reading data from d_crd array. For D attribute, generate a loop to iterate over every coordinate on this index. The size of this index is stored in $d\_pos[0]$. For S attribute, it only generates operations to load the coordinates from d_crd array. Besides generating loop structure code for each index, this step also updates three index variables ($vIdx_T$, $T \in \{A, B, C\}$) that will be used for inner-most computation. If the format attribute of an index (e.g., d) is D, i.e., d only appears in dense tensors, then $vIdx_T = vIdx_T \times d\_SIZE + arg$, where T denotes all dense tensors that contain index d, arg is the coordinate on index d (i.e., the argument of the generated loop for index d), and d_SIZE is index d's dimension size. If the format attribute of index d is sparse, this step handles sparse tensors and dense tensors separately. For sparse tensors T that contain index d, $vIdx_T = arg$, where arg is still the argument of the generated loop for index d. For dense tensors T that contain index d, $vIdx_T = vIdx_T + d\_crd[arg]$, where d_crd is the `crd` array of index d, and $d\_crd[arg]$ is the coordinate.

Step-III (Line 20) generates inner-most computation code to load values from $A[vIdx_A]$ and $B[vIdx_B]$, compute their product, and update $C[vIdx_C]$, after step-II generates $vIdx_T$ for tensor T ($T \in \{A, B, C\}$). More specifically, for sparse output code generation, the coordinate arrays of the sparse output can be inferred from the input tensors. If the dimension d is sparse in output tensor, the format of dimension d in

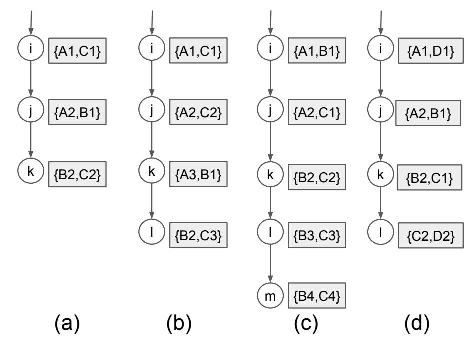


Fig. 7: Illustration of iteration graphs for (a) SpMM ($C_{ik} = A_{ij} * B_{jk}$) (b) TTM ($C_{ijl} = A_{ijk} * B_{kl}$) (c) $C_{jklm} = A_{ij} * B_{iklm}$ (d) $D_{il} = A_{ij} * B_{jk} * C_{kl}$

output follows the format of dimension d in the sparse input, so the coordinates in dimension d in the output is the same as the coordinates in dimension d in sparse input. For dense dimensions, we store the dimension size, which can be easily inferred from the input.

C. Input-dependent Optimization

The distribution of the nonzero entries in sparse tensors can significantly affect the performance. Reordering [33], [24] is an effective technique to optimize the memory access pattern caused by uneven data distribution. Different from existing compiler frameworks [17], [34] which apply reordering to iterations, we apply reordering to sparse matrices and tensors to improve data locality. In particular, we have integrated the-state-of-the-art reordering algorithm for tensors [24], called `LexiOrder`, in the compiler. This data reodering algorithm is implemented as an utility function in the runtime (`tensor_reorder()`) and called before the actual sparse computation in the TA dialect. The `LexiOrder` algorithm is built on top of the doubly lexical ordering algorithm [35] with some optimization techniques to advance its overall efficiency and availability on some concern cases. The basic idea behind this reordering algorithm is to sort a specific dimension (either rows or columns for matrices) in an iteration using the doubly lexical ordering algorithm and sort all dimensions in turn across iterations. The algorithm's objective is to cluster all nonzero entries around the diagonal to increase spatial and temporal locality.

D. Parallel Code Generation

For sequential execution the proposed compiler lowers the SCF dialect to the LLVM IR dialect and then to proper LLVM IR for assembly and linking. For parallel execution, instead, the SCF dialect is lowered to the `async` dialect (See Figure 1). In details, we developed a pass to lower `scf.for` loops to `scf.parallel` loops and the latter to the `async` dialect. The `async` dialect encapsulates the semantics of an asynchronous task-based parallel runtime in which computational tasks are spawn and asynchronously executed by parallel worker threads. Currently, MLIR supports a task continuation stealing approach in which the control is returned to the parent task after spawning. The dialect provides semantics primitives to synchronize the execution of tasks. Our compiler lowers those asynchronous task execution primitives to LLVM co-routines in LLVM IR. As Figure 8d shows, the MLIR asynchronous runtime introduces relatively low overhead during execution, which improves performance, especially for small computations.

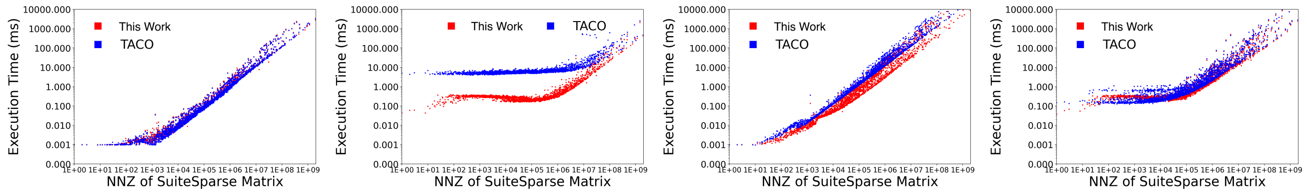
Name	Size	Nonzeros	Domain
NELL-1	2,902,330 x 2,143,368 x 25,495,389	143599552	Natural Language Processing
NELL-2	12,092 x 9184 x 28,818	76879419	Natural Language Processing
delicious-3d	532,924 x 17,262,471 x 2,480,308	140,126,181	Tags from Delicious website
flickr-3d	319,686 x 28,153,045 x 1,607,191	112,890,310	Tages from Flickr website
vast-2015-mc1-3d	165,427 x 11,374 x 2	26,021,854	Theme park attend event
Freebase-music[36]	23,344,784 x 223,344,784 x 166	99,546,551	Entries related with music in Freebase

TABLE I: Sparse tensor descriptions

VII. Evaluation

In this section, we evaluate the performance of automatically generated sparse kernels via our compiler. We also compare our results against TACO [30], a state-of-the-art tensor algebra compiler that performs source-to-source transformation from TACO DSL to sequential C++, shared-memory parallel OpenMP, and data-parallel CUDA code. For brevity, we evaluated the performance of selected benchmarks with a single storage format – matrices (CSR) and tensors (CSF), though our compiler can operate on other formats mentioned in Section IV. All results reported are the average of 25 runs.

We performed our experiments on a compute node equipped with two Intel Xeon Gold 6126 sockets running at 2.60GHz. Each CPU socket consists of 12 processing core (for a total of 24 cores). The system features 192 GB of DRAM memory. We use `clang 12.0` with optimization level `-O3` for all the experiments reported in this paper. We use as input datasets $2,833$ matrices and six tensors of different sizes and shapes chosen from the SuiteSparse Matrix Collection [37], the FROSTT Tensor Collection [38], and BIGtensor [36]. The SuiteSparse Matrix Collection is a growing dataset of sparse matrices in real-world applications. The dataset is widely used in the numerical linear algebra community for performance evaluation. The FROSTT Tensor Collection is a composition of open-source sparse tensor datasets from various data sources. The BIGtensor dataset is a tensor database that contains large-scale tensors for large-scale tensor analysis. Our input datasets represent the most important HPC domains in scientific computing, including chemistry, structural engineering, various linear solvers, computer graphics and vision, and molecular dynamics. We provide the description of the six tensors in Table I.



(a) sequential SpMV (b) parallel SpMV (c) sequential SpMM (d) parallel SpMM

Fig. 8: Sequential and parallel performance comparison with TACO for matrix operations.

A. *Sparse Tensor Operations*

We evalute the following sparse operations:

SpMV. The Sparse Matrix-times-Vector, $\mathbf{y} = \mathbf{X} \times \mathbf{v}$, is the multiplication of a sparse matrix $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2}$ with a dense vector $\mathbf{v} \in \mathbb{R}^{I_2}$. $y_{i_1} = \sum_{i_2=1}^{I_2} x_{i_1 i_2} v_{i_2}$.

SpMM. The Sparse Matrix-times-Matrix, $\mathbf{Y} = \mathbf{X} \times \mathbf{U}$, is the multiplication of a sparse matrix $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2}$ with a dense matrix $\mathbf{U} \in \mathbb{R}^{I_2 \times R}$. $y_{i_1 r} = \sum_{i_2=1}^{I_2} x_{i_1 i_2} u_{i_2 r}$.

SpTTV. The Sparse Tensor-Times-Vector (SpTTV) in mode n, $\mathcal{Y} = \mathcal{X} \times_n \mathbf{v}$, is the multiplication of a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ with a dense vector $\mathbf{v} \in \mathbb{R}^{I_n}$, along mode n. Given $n = 1$, $y_{i_2 i_3} = \sum_{i_1=1}^{I_1} x_{i_1 i_2 i_3} v_{i_n}$. This results in a two-dimensional $I_2 \times I_3$ tensor which has one less dimension.

SpTTM. The Sparse Tensor-Times-Matrix (SpTTM) in mode n, denoted by $\mathcal{Y} = \mathcal{X} \times_n \mathbf{U}$, is the multiplication of a sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ with a dense matrix $\mathbf{U} \in \mathbb{R}^{I_n \times R}$, along mode n. Mode-1 TTM results in a $R \times I_2 \times I_3$ tensor, and its operation is defined as $y_{r \cdots i_2 i_3} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 i_3} u_{i_n r}$. Note that R is typically much smaller than I_n in low-rank decompositions, typically $R < 100$.

SpMV and SpMM widely appear in applications from scientific computing, such as direct or iterative solvers [39], to data intensive domains [9], graph analytics [8]. SpTTV and SpTTM are computational kernels of popular tensor decompositions, such as the Tucker decomposition [40], tensor power method [41], for a variety of applications, including data analytics, numerical simulation, machine learning.

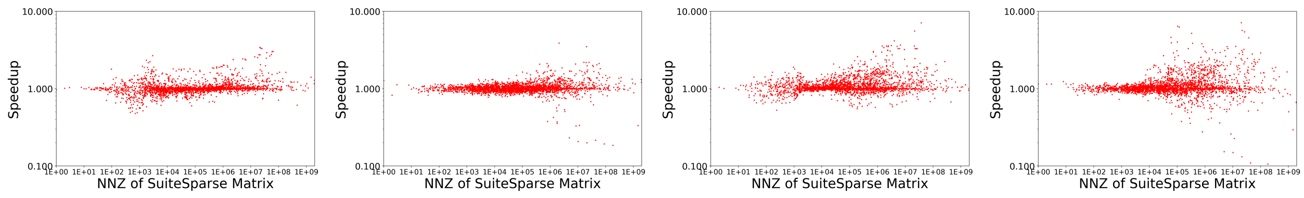
B. *Performance Evaluation*

SpMV and SpMM. We measured the performance of our compiler and TACO while running SpMV and SpMM with each of the 2,833 input matrices for sequential and parallel execution on multi-core architecture. We present the experimental results in Figure 8, where Our compiler and TACO are represented in red and blue dots, respectively. In the plot, the x-axis represents a matrix (2,833 matrices, ordered by increasing number of nonzeros) and the y-axis represents execution time (lower is better). As shown in Figure 8, in general, we achieves better performance than TACO.

For sequential execution, this work significantly outperforms TACO on average 2.29x, up to 6.26x for SpMM (Figure 8c) and on average 1.05x, up to 2.14x for SpMV (Figures 8a). To understand the performance difference better, we have investigated LLVM IR generated by our compiler and TACO and we notice that our compiler results in more optimized code with better SIMD (or vectorization) utilization and loop unrolling. For both in SpMM and SpMV, the utilization of many SIMD instructions in TACO is only half of that in the generated code by our compiler (e.g., TACO only uses 2 lanes while this work uses 4 lanes). Our compiler unrolls multiple loops by 8 while TACO unrolls them by 2. Although the generated LLVM IR for both SpMV and SpMM show similar differences, the effect of better vectorization and loop unrolling are more evident for larger computation (SpMM). The reason of why we generate more efficient code with better loop vectorization and unrolling strategies than TACO is that our compiler retains more semantic information during the progressive lowering. TACO only leverages the optimizations in LLVM and most semantic information is dropped when lowering from C++ to LLVM IR. In this work, the information is retained at the sparse dialect before lowering to LLVM IR, such as the dimensions of the indices, which allows us to leverage optimizations during the progressive lowering (e.g., constant propagation) for better code optimizations. This facilitates LLVM making better optimization decisions. These results highlight one of the major goals of MLIR and MLIR-based compilers: by leveraging higher-level semantics information and progressive lowering steps, it is possible to produce a more aggressive and higher-quality LLVM IR that might result in higher performance.

For parallel SpMV (Figure 8b), this work achieves an average of 20.92x speedup over TACO. Especially for small matrices, we outperform TACO by a significant margin, however, after further inspection, we realized that this performance difference is due to the overhead introduced by the underlying parallel runtime. Our compiler uses an asynchronous task-based programming model based on LLVM co-routines while TACO leverages OpenMP. For small computation, LLVM co-routines introduce less overhead than OpenMP threading (which is beneficial for larger parallel regions). As we can see from Figure 8d for SpMM, when there is enough computation for each OpenMP thread, the runtime overhead is amortized and our performance is similar to TACO's.

Reordering. By reordering data in memory, our compiler attempts to increase spatial and temporal locality to achieve higher performance. The plots in Figure 9 show our performance with data reordering as compared to original case (no data reordering). Figure 9 shows that, indeed, in many cases there is significant advantage of reordering data, with up to 3.41x (average 1.04x), 3.89x (average 1.03x), 7.12x (average 1.12x), and 7.14x (average 1.13x) for SpMV sequential, SpMV



(a) sequential SpMV-lexi (b) parallel SpMV-lexi (c) sequential SpMM-lexi (d) parallel SpMM-lexi

Fig. 9: Performance when employing Lexi ordering with respect to original case.

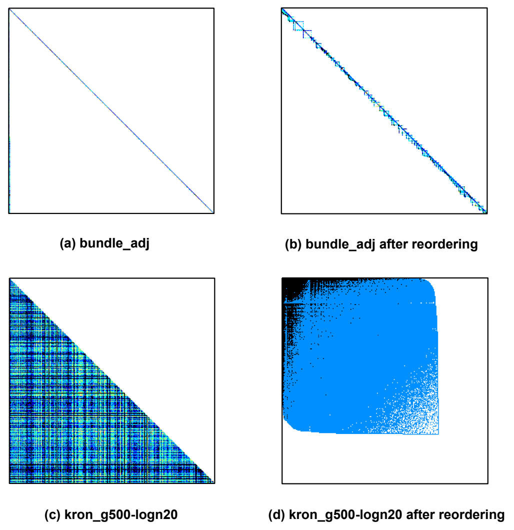


Fig. 10: Visualization of matrices with and without reordering

parallel, SpMM sequential, and SpMM parallel, respectively. However, we also note that there might be significant performance degradation, especially for parallel execution. We further analyzed the reasons for this disparity and identified load imbalance as the primary source of performance degradation. Our reordering algorithm clusters nonzeros on the top-left corner of sparse matrices. In an ideal case, after reordering the nonzeros are distributed around the matrix diagonal.

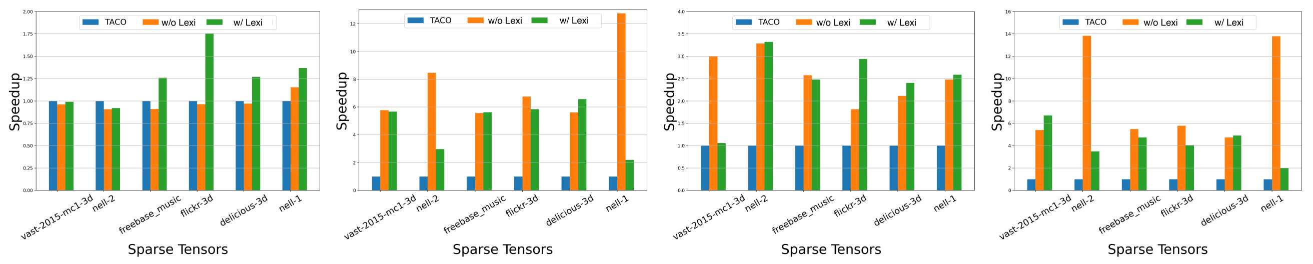
We have evaluated two cases for SpMM in which the number of nonzeros for the selected matrices are similar but the performance of reordering is significantly different from each other. Figure 10(a)-(b) shows a case in which reordering results in high performance improvements. In this case, the nonzero elements originally around the first column are distributed around the diagonal. Figure 10(c)-(d), instead, shows a case in which reordering reduces performance. In this case, the nonzeros are clustered around the top-left corner, thus threads that operate on the top rows have more work to perform compared to threads that operate on the bottom rows, which results in load imbalance and performance degradation.

TTV and TTM. We also compare our work with TACO on TTV and TTM with six sparse tensors for sequential and parallel execution with reordering optimization on and off. Figure 11 illustrates the experimental results. TACO is not able to generate parallel code if the output tensor is stored in sparse format, thus the results in the figure for parallel execution are with respect to sequential execution of the TACO benchmarks. For sequential TTV, our performance is comparable to TACO. With reordering, we achieve better performance on four out of six sparse tensors. For parallel TTV, our performance is significantly better than TACO with up to 12.5× and on average 8× speedup. With reordering, the performance is degraded on five of six sparse tensors except for delicious-3d. As for the case of SpMV and SpMM, we observed similar load imbalance issues. For sequential TTM, our work performs better than TACO with up to 3.3× and on average 2.53× speedup. With reordering, we achieves better performance on three out of six sparse tensors. For parallel TTM, we performs significantly better than TACO with up to 13.9× and on average 8.13× speedup. With reordering, the performance is degraded on five of six sparse tensors except for vast-2015-mc1-3d.

Our results show that reordering tensors have a significant (positive or negative) impact on performance, more than for matrices. One possible reason is that the `LexiOrder` algorithm reorders all dimensions of data simultaneously, which means the data locality is the best when accessing all the dimensions in conjunction, as in conjunction. The sparse tensor operation MTTKRP [24] follows this behavior to gain a good performance speedup. However, this does not mean that the indices in every dimension get good locality when accessing the vector or matrix in TTV or TTM, potentially leading to low performance. We will investigate alternative reordering algorithms and adaptive methods in future work.

Summary. We summarized our performance compared with TACO in Table II. This work outperforms the state-of-the-art sparse compiler TACO because of three aspects:

- Better SIMD vectorization and loop unrolling for sequential execution: In TACO, most semantic information is dropped when lowering from C++ to LLVM IR. In this work, high-level semantic information is retained at the sparse dialect before lowering to LLVM IR, which allows us to leverage additional optimizations in the LLVM back-end.
- Better runtime for parallel execution: our compiler uses an asynchronous task-based programming model based on LLVM co-routines, while TACO uses OpenMP.
- Input-dependent optimizations (data reordering): TACO does not perform any input-dependent optimizations while this work reorganizes data for better load balancing.



(a) sequential TTV (b) parallel TTV (c) sequential TTM (d) parallel TTM

Fig. 11: Performance of sequential and parallel execution with respect to TACO for tensor operations.

	SpMV				SpMM				TTV				TTM			
	Without Lexi.		With Lexi.		Without Lexi.		With Lexi.		Without Lexi.		With Lexi.		Without Lexi.		With Lexi.	
	Seq.	Para.	Seq.	Para.	Seq.	Para.	Seq.	Para.	Seq.	Para.	Seq.	Para.	Seq.	Para.	Seq.	Para.
Avg.	1.05x	20.92x	1.09x	21.54x	2.29x	1.21x	2.56x	1.36x	0.99x	8x	1.26x	4.8x	2.53x	8.13x	2.45x	4.2x
Upto	2.14x	194.8x	3.60x	308.8x	6.26x	6.39x	12.2x	11.69x	1.18x	12.5x	1.75x	6.6x	3.3x	13.9x	3.35x	6.7x

TABLE II: Performance summary with respect to TACO

VIII. Related Work

Compiler for Tensor Algebra. Compiler techniques have been used to drive tensor algebra computation [42], [30]. TCE [42] is a compiler optimization framework that focuses on dense tensor contraction operations in quantum chemistry. Similarly, COMET [18] proposes a domain-specific compiler for dense tensor algebra computations, focusing on computational chemistry kernels in NWChem. The COMET compiler is built on top of the MLIR infrastructure and it reformulates tensor contraction operations via equivalent transpose-transpose-GEMM-transpose(TTGT) expressions to take advantage of optimized GEMM kernels. In the sparse domain, TACO [30] is the state-of-the-art sparse source-to-source compiler that generates efficient code for given tensor algebra expressions with different storage formats, targeting multi-core architecture and NVIDIA GPUs. TACO translates DSL operations to C++/OpenMP/CUDA using computational templates. Adding support for a new architecture, in general, requires a new and separate code generation path, which is a tedious effort. Our approach is more flexible and extensible and allows sharing dialects and optimizations across DSLs and heterogeneous architectures. One could envision using TACO DSL on top of our multi-level IR stack, but not vice-versa. Different from existing works, we develop a high-performance sparse tensor algebra compiler in multi-level IR, which supports both serial and parallel code generation and more importantly enables interoperability, and composability, and better portability for future architectures.

Domain-specific Libraries for Tensor Algebra. There have been a collection of tensor algebra libraries developed. FLAME [28] is a library aiming for the derivation and implementation of tensor algebra operations on CPUs. Later, serial linear algebra libraries are extended to run on distributed parallel systems [29]. On the other hand, these libraries are extended to support sparse tensor algebra operations using different sparse tensor formats [43]. Tensor algebra libraries favor scientific computing and are widely utilized in scientific application development. By contrast, this work transparently implements tensor algebra algorithms per se and can compile most types of sparse tensor formats and automatically generate efficient code.

Tensor Algebra Optimization. Plenty of algorithms [11], [44] exist for data reordering to optimize tensor algebra with respect to distinct tensor formats for different tensor operations and heterogeneous architectures. Kjolstad et al. [25], [7] reorder loops of tensor algebra computations to improve the data locality. Smith et al. [11] use reordering to enable high-performance tensor factorization operations. Yang et al. [44] identify an efficient memory access pattern for high-performance SpMM operations through merge-based load balancing and row-major coalesced memory access.

IX. Conclusion

In this work, we present a high-performance sparse tensor algebra compiler in multi-level IR and a high-productive DSL to support complex tensor applications. Our DSL enables high-level programming abstractions that resemble the familiar Einstein notation to express tensor algebra operations. The proposed compiler is based on the MLIR framework, which allows us to build portable, adaptable, and extensible compilers. It provides a novel and efficient code generation algorithm, which supports most tensor storage formats through an internal storage format based on four dimension attributes. Furthermore, we incorporate a data reordering algorithm to increase the data locality and load balancing. Our compiler outperforms TACO, a state-of-the-art sparse tensor algebra compiler, with up to 20.92x, 6.39x, and 13.9x performance improvement for SpMV, SpMM, and TTM computations, respectively. In future work, we plan to extend our compiler to support heterogeneous architectures and to explore alternatives reordering schemes. Finally, we plan to release our code as an open source to the broader community.

References

[1] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *2008 Eighth IEEE international conference on data mining*. IEEE, 2008, pp. 363–372.

[2] S. Rendle, L. Balby Marinho, A. Nanopoulos, and L. Schmidt-Thieme, "Learning optimal ranking with tensor factorization for tag recommendation," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 727–736.

[3] E. Acar, C. Aykut-Bingol, H. Bingol, R. Bro, and B. Yener, "Multiway analysis of epilepsy tensors," *Bioinformatics*, vol. 23, no. 13, pp. i10–i18, 2007.

[4] G. Bouchard, J. Naradowsky, S. Riedel, T. Rocktäschel, and A. Vlachos, "Matrix and tensor factorization methods for natural language processing," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing: Tutorial Abstracts*, 2015, pp. 16–18.

[5] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1933–1936, 2017.

[6] Y. Zhang, M. Chen, S. Mao, L. Hu, and V. C. Leung, "Cap: Community activity prediction based on big data analysis," *IEEE Network*, vol. 28, no. 4, pp. 52–57, 2014.

[7] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[8] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive autotuner for sparse matrix-vector multiplication," in *Proceedings of the conference on Programming language design and implementation*, 2013, pp. 117–126.

[9] X. Zhang, Y. Zhang, X. Sun, F. Liu, S. Liu, Y. Tang, and Y. Li, "Automatic performance tuning of spmv on gpgpu," *HPC Asia, Kaohsiung, Taiwan, China*, pp. 173–179, 2009.

[10] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin, "Efficient and scalable computations with sparse tensors," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–6.

[11] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 61–70.

[12] J. Li, J. Sun, and R. Vuduc, "Hicoo: hierarchical storage of sparse tensors," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 238–252.

[13] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–25.

[14] A. J. C. Bik and H. A. G. Wijshoff, "Compilation techniques for sparse matrix computations," ser. ICS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 416–424.

[15] S. Thibault, L. Mullin, and M. Insall, "Generating indexing functions of regularly sparse arrays for array compilers," 11 1996.

[16] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276493

[17] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "taco: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 943–948.

[18] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "Comet: A domain-specific compilation of high-performance computational chemistry," in *Workshop on Languages and Compilers for Parallel Computing (LCPC'20)*. Springer.

[19] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[20] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–26, 2020.

[21] J. B. White and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *Proceedings Fourth International Conference on High-Performance Computing*. IEEE, 1997, pp. 66–71.

[22] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–11.

[23] D. R. Kincaid, T. C. Oppe, and D. M. Young, "Itpackv 2d user's guide," Texas University. Center for Numerical Analysis, Tech. Rep., 1989.

[24] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 227–237.

[25] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 180–192.

[26] "National institute of standards and technology," http://math.nist.gov/MatrixMarket/ formats.html, 2013.

[27] S. Smith, J. L. Jee W. Choi, R. Vuduc, J. Park, X. Liu, and G. Karypis, "Frostt file formats." http://frostt.io/tensors/file- formats.html, 2017.

[28] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. Van De Geijn, "Flame: Formal linear algebra methods environment," *ACM Transactions on Mathematical Software (TOMS)*, 2001.

[29] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *Transactions on Mathematical Software*, 2013.

[30] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *The ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[31] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[32] R. F. Boisvert, R. F. Boisvert, and K. A. Remington, *The matrix market exchange formats: Initial design*. US Department of Commerce, National Institute of Standards and Technology, 1996, vol. 5935.

[33] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 376–388.

[34] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of The Int. Conference on Parallel Architectures and Compilation*, 2016, pp. 299–312.

[35] R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973–989, 1987.

[36] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *IEEE International Conference on Data Engineering (ICDE)*, 2015.

[37] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 2011.

[38] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, "Frostt: The formidable repository of open sparse tensors and tools," 2017.

[39] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[40] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[41] A. Anandkumar, R. Ge, and M. Janzamin, "Analyzing tensor power method dynamics in overcomplete regime," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 752–791, 2017.

[42] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *The Journal of Physical Chemistry A*, 2003.

[43] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, "A massively parallel tensor contraction framework for coupled-cluster computations," *Journal of Parallel and Distributed Computing*, 2014.

[44] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," in *European Conference on Parallel Processing*. Springer, 2018.