Compiler Support for Sparse Tensor Computations in MLIR

AART BIK, PENPORN KOANANTAKOOL, TATIANA SHPEISMAN,
NICOLAS VASILACHE, and BIXIA ZHENG, Google, USA
FREDRIK KJOLSTAD, Stanford University, USA

Sparse tensors arise in problems in science, engineering, machine learning, and data analytics. Programs that operate on such tensors can exploit sparsity to reduce storage requirements and computational time. Developing and maintaining sparse software by hand, however, is a complex and error-prone task. Therefore, we propose treating sparsity as a property of tensors, not a tedious implementation task, and letting a sparse compiler generate sparse code automatically from a sparsity-agnostic definition of the computation. This article discusses integrating this idea into MLIR.

CCS Concepts: • **Software and its engineering → Compilers**; **Domain specific languages**; • **Mathematics of computing → Mathematical software;**

Additional Key Words and Phrases: Compilers, sparse data structures, tensor algebra, machine learning

ACM Reference format:
Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Arch. Code Optim.* 19, 4, Article 50 (September 2022), 25 pages.
https://doi.org/10.1145/3544559

1 INTRODUCTION

Vectors, matrices, and their higher-dimensional generalization into tensors that contain many zero elements are called sparse tensors. Sparse tensors arise in a wide range of problems in science, engineering, machine learning, and data analytics. Programs that operate on such tensors can exploit sparsity to reduce both storage requirements and computational time by only storing the nonzero elements and skipping computations with a trivial outcome (such as `x+0=x` and `x*0=0`). This exploitation comes at a cost, though, since developing and maintaining sparse software by hand is a rather complex and error-prone task. Therefore, we propose treating sparsity merely as a property of tensors, not a tedious implementation task, and letting a **sparse compiler** generate sparse code automatically from a sparsity-agnostic definition of the computation.

In this article, we describe our experience integrating compiler support for sparse tensor computations into the MLIR open-source compiler infrastructure [56] and the opportunities and challenges that arise. The idea of making sparsity a property composes well with MLIR's ease of defining multi-level abstractions and progressively lowering transformations that continuously operate at the most appropriate level of abstraction.

Authors' addresses: A. Bik (corresponding author), P. Koanantakool, T. Shpeisman, N. Vasilache, and B. Zheng, Google 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA; email: ajcbik@google.com; F. Kjolstad, Stanford University 353 Jane Stanford Way Stanford CA 94305, USA.

1544-3566/2022/09-ART50
https://doi.org/10.1145/3544559

Our sparse compiler support in MLIR consists of a new sparse dialect that provides the attributes, types, operations, and transformations that are required to make sparse tensor types first-class citizens in MLIR. The sparse dialect forms a bridge between high-level operations on sparse tensor types and low-level operations on sparse storage formats that avoid redundant work by only storing and operating on nonzero elements. Central to sparse tensor types is an attribute that defines the desired way of storing sparse tensors. For example, the following MLIR representation of a dense matrix multiplication `C[i,j] = A[i,k] * B[k,j]`, where type `tensor<mxnxf64>` denotes a `m` × `n` double-precision matrix

```
%C = linalg.matmul ins(%A, %B: tensor<2x4xf64>, tensor<4x8xf64>) -> tensor<2x8xf64>,
```

changes into a sparse operation by *merely* adding a sparse attribute `#Sparse` to the tensor type of one or more operands (the precise contents of such a sparse attribute will be discussed later in the article), as done below for matrix `A`:

```
%C = linalg.matmul ins(%A, %B: tensor<2x4xf64, #Sparse>, tensor<4x8xf64>) -> tensor<2x8xf64>
```

By making sparsity an optional property of the tensors, we avoid the proliferation of specialized routines for such operations. Instead, after introducing the sparse tensor attribute, compiler transformations take care of lowering the operation to imperative constructs and sparse storage formats that only store and iterate over nonzero elements to perform the matrix multiplication. Using a different sparse attribute for matrix `A`, or adding sparse attributes to the tensor types of matrices `B`, `C`, or both, prompts the compiler to generate completely different sparse code. Since programmers merely annotate sparse tensor types and leave the tedious implementation task to the sparse compiler, this approach greatly simplifies sparse code development compared to traditional hand-written approaches. Now, a single sparsity-agnostic description can be mapped into a wide range of sparse implementations, each tailored to specific instances of the same problem.

The specific contributions of our article are

- a description of an industrial sparse tensor algebra compiler as an MLIR dialect,
- composition of the sparse dialect with other MLIR dialects and transformations,
- two use cases of the sparse tensor algebra compiler, and
- empirical results showing that composition leads to good performance.

The rest of this article is organized as follows. Section 2 provides preliminaries related to sparse tensors. Sections 3 and 4 dive into the design and implementation details of adding sparse compiler support to the MLIR compiler infrastructure. Section 5 explores various ways to use the new support in MLIR. An experimental validation of the sparse compiler follows in Section 6. Related work is discussed in Section 7. Finally, conclusions and future plans appear in Section 8.

2 SPARSE PRELIMINARIES

This section provides preliminaries on sparse tensors, sparse storage formats, and sparse compilers.

2.1 Sparse Tensors and Storage Formats

A **tensor** is a d-dimensional generalization of one-dimensional vectors and two-dimensional matrices. If many elements in the tensor are zero, the tensor is called a **sparse tensor**, which is a situation that arises often in problems in science, engineering, machine learning, and data analytics. In contrast, a tensor without this property is called a **dense tensor**. One can furthermore distinguish between *unstructured* sparse tensors that have no discernible nonzero structures and *structured* sparse tensors that have particular nonzero structures, such as tensors with nonzero elements confined within blocks, bands, diagonals, or borders, or with certain statistical properties on the distribution of nonzero elements.

Programs that operate on sparse tensors can take advantage of the sparsity to reduce *storage requirements* by only storing the nonzero elements and *computational time* by skipping trivial operations (such as `x+0=x` and `x*0=0`). How to effectively exploit sparse vectors and matrices has been well studied in the past for linear algebra problems; see, for example, [2, 20, 22, 26–28, 30, 32, 37, 59, 60, 68, 71, 85, 90, 97]. The growing popularity of deep learning and big data has sparked a similar interest in studying how machine learning kernels can take advantage of sparse tensors [18, 48, 50, 51, 76, 79, 84].

Many ways of storing sparse tensors have been proposed [14, 15, 39, 44, 47, 57, 65, 72, 74, 75, 80, 87], and which of these storage formats is most effective in terms of minimizing storage as well as computation depends on the peculiarities of the nonzero structure, the operations to be performed, and the target architecture. All sparse storage formats consist of a **primary storage** that stores the numerical values of the nonzero elements and some **overhead storage** needed to map those values to their tensor coordinates, in order to reconstruct the enveloping tensor from these values. Sometimes, even some zero elements are stored explicitly to accommodate simpler structured formats or to avoid costly removals of nonzero elements that become zero during the computation. Such explicit zeros, however, do not impact the correctness of tensor operations.

Well-known examples of sparse storage formats include **coordinate format (COO)** [75], **hierarchical coordinate (HiCOO)** [57], **compressed sparse row/column (CSR and CSC)** [87], **doubly compressed sparse row/column (DCSR and DCSC)** [15], **compressed sparse fiber (CSF)** [80], block [44], band, **diagonal (DIA)**, **jagged diagonal (ELL)** [47, 65], **adaptive linearized storage (ALTO)** [39], and hash maps. Although several examples appear later in the article, for an in-depth survey of sparse storage formats, we must refer to the literature.

2.2 Sparse Compilers

Writing effective sparse code is a time-consuming and error-prone task, further complicated by the large number of possible combinations of storage formats, nonzero structures, operations, and target architectures. Due to the complexity, programmers instead usually restrict themselves to hand-optimizing a small set of library methods for specific operations and storage formats and building larger sparse programs by composing available library methods. This approach may lead to sub-optimal performance when costly data structure conversions are needed in between library methods or, worse, sub-optimal asymptotic complexity when many avoidable intermediate results produced by one library method are nullified after multiplication by zeros in the next method.

Rather, we would like to *treat sparsity merely as a property*, not a tedious implementation task, and let a **sparse compiler** generate sparse code automatically from a sparsity-agnostic definition of the computation. This idea of keeping sparsity completely transparent to the programmer was pioneered in the *MT1 sparse compiler* for linear algebra [7–10] and later formalized and generalized to tensor algebra in *TACO (Tensor Algebra Compiler)* [48–51]. With the figurative push of a button, a sparse compiler can convert a single sparsity-agnostic description into a wide range of sparse implementations, each tailored to specific instances of the same problem with tensors stored in different data structures. This automatic approach not only enables non-expert programmers to generate sparse code quickly but also enables expert programmers to explore the full space of possible sparse implementations.

3 A SPARSE COMPILER IN MLIR

This section discusses some of our design considerations from implementing a sparse compiler in the MLIR compiler infrastructure, which is part of the LLVM open-source project.

3.1 MLIR Compiler Infrastructure

The MLIR open-source project [55, 56] provides an extensible infrastructure for building compilers for domain-specific languages. The infrastructure provides a way to specify new intermediate representations through **dialects** together with transformations on these representations. Transformations can be written as compositions of orthogonal localized match and rewrite primitives. These are often decomposed further into **rewriting rules** when applied within a dialect and **lowering rules** when converting from a higher-level dialect to a lower-level dialect. Each dialect can define custom attributes, types, and operations. Throughout the compilation, separate dialects can co-exist to form a hybrid representation of a program.

The ability to progressively lower to dialects closer and closer to the target hardware during the compilation process, together with an intuitive transformation mechanism, has made MLIR a popular compiler infrastructure for domain-specific languages that need to bridge large semantic gaps, such as compiling for machine learning. In this article, we rely on several MLIR dialects, whose relationships are shown in Figure 1. The dialects are briefly described below.

Linalg. The `Linalg` dialect, inspired by the tensor index notation found in tensor comprehensions [89], provides high-level primitives that know how to decompose themselves into loops. A key design of the current work was to leave the operational semantics of the dialect unchanged, except that it now supports dense and sparse data types. We assume that high-level kernels written in, e.g., Tensorflow, JAX, or Python are presented to the sparse compiler as `Linalg` operations, possibly after lowering from higher-level dialects.

Tensor. The `Tensor` dialect operations manipulate an abstract tensor type for which the compiler infrastructure has not yet decided on a representation in memory. Small tensors with static sizes may be directly promoted to constants. When tensors are large enough and/or have dynamic sizes, they are assigned memory storage through a **bufferization** process.

SparseTensor. The `SparseTensor` dialect, discussed in more detail in Section 4, is new to this work and provides the attributes, types, operations, and transformations that are required to make sparse tensor types first-class citizens within the MLIR compiler infrastructure. The dialect provides a bridge between high-level operations on these sparse tensor types and low-level operations that only store and operate on nonzero elements to avoid performing redundant work. It co-exists with the `Tensor` dialect to provide a logical separation between purely dense and sparse functionality.

SCF. The structured control-flow `SCF` dialect provides operations that represent looping and conditionals, e.g., regular `for` and `while` loops as well as an `if` conditional. This is structured at a higher level of abstraction than **control-flow graphs (CFGs)**. Notably, loop operations may yield SSA values and compose well with other operations and dialects. This dialect is used by the sparse compiler to decompose sparse `Linalg` operations into explicit imperative constructs.

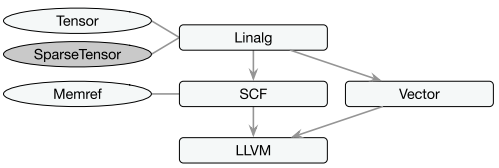

Fig. 1. The MLIR dialects involved in sparse compilation. Ovals contain data, rectangles contain code, lines indicate data that is used by a dialect, and arrows indicate lowering paths. Although sparse compilation only adds the `SparseTensor` dialect, it reuses `Linalg` to define its operations and `SCF` and `Vector` as its targets.

Memref. The `Memref` dialect introduces the `memref` data type, which is the main representation for memory buffers in MLIR and the entry point to the side-effecting memory-based operations. The `memref` data type also provides an unsurprising ABI to interoperate with external C code and serves as a bridge for calling libraries from MLIR code generation. Dense tensors undergo a relatively straightforward bufferization into memory buffers (viz. multi-dimensional arrays). The bufferization of sparse tensors is more elaborate, as will be seen later in this article.

Vector. The `Vector` dialect provides an architectural-neutral representation of vector operations that is eventually progressively lowered into machine-specific SIMD instructions.

3.2 Sparse Compiler Design Philosophy

The growing popularity of the MLIR open-source project together with the growing interest in sparse tensors in machine learning and data analytics prompted adding compiler support for sparse tensor computations to the MLIR compiler infrastructure. The idea of making sparsity a property composes well with MLIR's ease of defining multi-level abstractions and progressively lowering transformations that continuously operate at the most appropriate level of abstraction. The *north star vision* of this project consists of providing an excellent, reusable sparse ecosystem to academia and industry that is based on first principles. To this end, emphasis was put on introducing sparse tensor types as first-class citizens into MLIR. Furthermore, a first *reference implementation* provides a fully functional implementation of this sparse ecosystem against which future versions can be compared. The long-term north star vision together with the shorter-term first reference implementation enables researchers to independently start developing MLIR front-ends for sparse array languages as well as new MLIR transformations that implement improved or even alternative sparse compilers. Our hope is that the open-source nature of MLIR will work both ways, that is, benefit researchers that are new to the sparse domain as well as solicit useful contributions from experts in the open-source community at large.

4 THE SPARSE TENSOR DIALECT

Sparse tensor support in MLIR mostly resides within a new `SparseTensor` dialect, which provides the attributes, types, operations, and transformations that are required to make sparse tensor types first-class citizens within the MLIR compiler infrastructure. The dialect forms a bridge between high-level operations on sparse tensor types and low-level operations on the actual sparse storage formats that only store and operate on nonzero elements to avoid performing redundant work.

4.1 Sparse Tensor Attributes and Types

Many choices are possible for the sparse tensor type specification itself, varying from simply annotating a tensor type as sparse to providing detailed information on the nonzero structure or other characteristics. We support a TACO-flavored mechanism of annotating sparse tensors [48], since this provides an elegant and powerful way of specifying a large number of different sparse storage formats. In true MLIR fashion, however, the type specification has been kept extensible to allow for more advanced annotations in the future.

Central to the chosen annotation mechanism is a sparse attribute that defines the desired way of storing each sparse tensor by encoding (1) *per-dimension level types* of either `dense` or `compressed`, (2) a *dimension ordering*, and (3) *bit widths* for pointers and indices.

```
#SparseTensor = #sparse_tensor.encoding<{
  dimLevelType = [ "dense", "compressed", ... ],
  dimOrdering = affine_map<(i,j,k,...) -> (i,j,k,...)>,
  pointerBitWidth = ...,
  indexBitWidth = ...,
}>
```

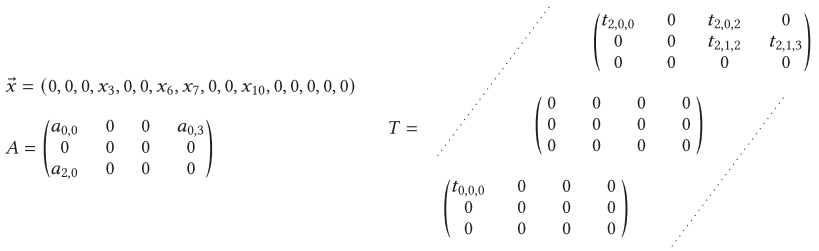

Fig. 2. Example tensors. We assume tensors use zero-based lexicographical indexing, e.g., a three-dimensional tensor T starts with element $t_{0,0,0}$ with, respectively, layer, row, and column index 0.

Sparse tensor types are obtained by annotating the built-in tensor types of MLIR with sparse format attributes as illustrated below:

```
tensor<..., #SparseTensor>
```

The attribute indicates that when the d-dimensional tensor is lowered into an actual storage scheme during bufferization, rather than selecting a straightforward `memref` buffer used for dense tensors, the sparse tensor is lowered into a compact sparse storage scheme that *conceptually* consists of two integral jagged arrays `pointers[`d`][*]` and `indices[`d`][*]`, where each entry in the first dimension stores, respectively, the pointer and index arrays of a single tensor dimension and a one-dimensional values array `values[*]` that provides sufficient space for all stored elements.

Consider, for example, the sparse vector $\vec{x}$ of Figure 2 together with the following sparse attribute and sparse vector type shown at the left:

```
#SparseVector= #sparse_tensor.encoding<{
   dimLevelType = [ "compressed" ]
}>
tensor<16xf64, #SparseVector>
```

pointers[0]:	0	4		
indices[0]:	3	6	7	10
values:	x_3	x_6	x_7	x_{10}

This type results in the compressed format shown at the right. Contents of `pointers[0][0]` and `pointers[0][1]` denote that elements appear at range `[0,4)` in the indices and values arrays. There, `indices[0][*]` and `values[*]` provide, respectively, the indices and values of nonzero elements. Memory is saved by only storing 10 values (4 doubles primary and 6 integers overhead storage) rather than the 16 consecutive doubles that would result in the dense case.

For matrices, CSR is defined with dimension level type `dense` for the first dimension and `compressed` for the second dimension, as follows. When absent, the dimension ordering defaults to lexicographical index order, thus, row-wise order in this case.

```
#CSR = #sparse_tensor.encoding<{
  dimLevelType = [ "dense", "compressed" ]
}>
tensor<3x4xf64, #CSR>
```

pointers[1]:	0	2	2	3
indices[1]:	0	3	0	
values:	$a_{0,0}$	$a_{0,3}$	$a_{2,0}$	

Then, the sparse matrix A of Figure 2 maps to the shown compressed format, which has an implicit dense first dimension of size 3 and only uses the second dimension of the jagged arrays. The nonzero values and corresponding column coordinates in each row $0 \leq$ `i` $<$ 3 can be found in the indices and values arrays at range `pointers[1][i]` up to `pointers[1][i+1]`. There, `indices[1][*]` and `values[*]` provide, respectively, column indices and values. Note that the number 2 is repeated in the pointers array because the second row of the matrix A is empty.

The storage format can be made doubly compressed using `compressed` for both dimension level types. In addition, an explicit *dimension ordering* can be used to enforce column-wise storage over the default lexicographical index order storage, as illustrated below with the definition of DCSC.

```
#DCSC = #sparse_tensor.encoding<{
  dimLevelType = [ "compressed", "compressed" ],
  dimOrdering  = affine_map<(i,j) -> (j,i)>
}>
tensor<3x4xf64, #DCSC>
```

pointers[0]:	0	2	
indices[0]:	0	3	
pointers[1]:	0	2	3
indices[1]:	0	2	0
values:	$a_{0,0}$	$a_{2,0}$	$a_{0,3}$

This type maps to the shown column-wise sparse storage format for the example matrix A. The contents in `pointers[0][0]` and `pointers[0][1]` denote that range `[0,2)` is used in the next arrays to store two columns. There, `indices[0][*]` denote that only column 0 and column 3 contain nonzero elements. The next level provides a similar structure within each stored column.

The attribute easily generalizes to tensor or arbitrary dimensions, as illustrated below:

```
#SparseTensor = #sparse_tensor.encoding<{
  dimLevelType = [ "compressed",
                   "compressed",
                   "compressed" ]
}>
tensor<3x3x4xf64, #SparseTensor>
```

pointers[0]:	0	2			
indices[0]:	0	2			
pointers[1]:	0	1	3		
indices[1]:	0	0	1		
pointers[2]:	0	1	3	5	
indices[2]:	0	0	2	2	3
values:	$t_{0,0,0}$	$t_{2,0,0}$	$t_{2,0,2}$	$t_{2,1,2}$	$t_{2,1,3}$

Given the 3-dimensional tensor T of Figure 2, this attribute results in the compressed data structure shown at the right. Stored elements appear in the default lexicographical index order. Contents in `pointers[0][0]` and `pointers[0][1]` reveal two filled layers, with `indices[0][*]` specifying layer 0 and 2 in particular. The next level `pointers[1][*]` shows that the matrix in the first stored layer has one filled row and the matrix in the second stored layer has two filled rows. The last level provides structure within each compressed row.

In addition to the per-dimension level types and optional dimension ordering, the sparse attribute can also specify alternative bit-widths ranging from 8 to 64 for pointers and indices. Narrow bit-widths can be used to reduce overhead storage requirements, provided that the widths still suffice to store the maximum possible integral pointer and index value throughout the storage format. Given the two choices for each per-dimension level type, all possible permutations for the dimension ordering, and the four choices for both bit-widths, this relatively simply sparse tensor type encoding already supports $2^d \cdot d! \cdot 16$ different ways of storing a single d-dimensional tensor. Many of the resulting storage formats directly correspond to common data structures found in the literature for storing unstructured sparse tensors, or sparse tensors with a simple structure along bands. Block sparsity [44] can be expressed as well with these annotations using a higher-dimensional structure together with a mapping back to the original lower-dimensional indices. In addition, as stated before, the encoding has been kept extensible to allow for specifying even more sparse storage formats in the future. For example, with just a few additional dimension level types, many more formats can be expressed, including COO, DIA, ELL, and hash maps, as discussed in detail in [18], as well as uncommon storage formats previously unexplored in research.

4.2 Sparse Tensor Operations

With sparse tensor types as first-class citizens, *any* MLIR tensor operation can be made sparse by simply annotating the tensor types of the operands. For example, the `Linalg` dialect representation in MLIR of a dense matrix multiplication operation `C[i,j] = A[i,k] * B[k,j]` looks as follows, where the ? in the tensor type denotes a runtime dimension size:

```
%C = linalg.matmul ins(%A, %B: tensor<?x?xf64>, tensor<?x?xf64>) -> tensor<?x?xf64>
```

This representation changes into a sparse kernel by *merely* adding sparse attributes to the tensor types of the operands, as shown below by storing all matrices in CSR format:

```
%C = linalg.matmul ins(%A,%B: tensor<?x?xf64,#CSR>,tensor<?x?xf64,#CSR>) -> tensor<?x?xf64,#CSR>
```

As such, the SparseTensor dialect only needs to provide a few operations specific to sparse tensor types: (1) materialization operations, (2) conversion operations, and (3) operations that support progressive lowering. Below, we briefly explore each category.

Materialization operations produce or consume tensor value in SSA form. For example, the new operation reads a sparse tensor from some source.

```
%0 = sparse_tensor.new %source : !Source to tensor<?x?x?xf32, #SparseTensor>
```

The source is typically a file in an external format provided by the Matrix Market [12] (a popular sparse matrix repository and successor of earlier sets like the Harwell-Boeing Sparse Matrix Collection [29] and SPARSKIT [73]), SuiteSparse [24, 25], HaTen2 [43], or FROSTT [78] (repository of sparse tensors). Reading from file assumes a target platform that supports a file system, but other more architecturally neutral sources are possible as well, such as initializing a sparse tensor from code or through some API from a memory-resident format owned by an external library. The init operation materializes an uninitialized tensor as SSA value into a computation. Conversely, the out operation, illustrated below, outputs a sparse tensor to some destination, typically by writing an external format like FROSTT to a file if the target architecture provides a file system. The operation is kept general, however, to support other kinds of destinations in the future as well.

```
sparse_tensor.out %0, %dest : tensor<?x?x?xf32, #SparseTensor>, !Destination
```

Since MLIR is strongly typed, built-in verification rejects implicit type casts between dense and sparse tensor types or between differently annotated sparse tensor types, since most of such type conversions would incur nontrivial costs. Instead, the conversion operation convert operation is used to make such casts explicit. An example of converting a 10×10 sparse matrix in CSR format to CSC format with dynamic dimensions is shown below.

```
%to = sparse_tensor.convert %from : tensor<10x10xf64, #CSR> to tensor<?x?xf64, #CSC>
```

In the first reference implementation, conversion operations between sparse formats are implemented by converting to and from an intermediate coordinate scheme, which avoids the quadratic trap of implementing all possible direct conversions. In the longer term, however, this approach can be replaced by more advanced schemes, at least in special cases, such as proposed in [19]. The convert operation also converts between dense tensor types and sparse tensor types. Although not practical for many real-world sparse tensors, where storage requirements of an enveloping dense tensor would be prohibitive, such conversions are useful to manipulate smaller datasets in tests. An example of initializing a sparse matrix from a compile-time constant that uses a COO-flavored mechanism of specifying the indices and values of all nonzero elements is shown below. Although, perhaps somewhat surprisingly, the type of SSA value %m is actually a *dense* 10×8 matrix, the subsequent conversion ensures that the *sparse* matrix in %s is initialized directly from the nonzero elements, without ever fully materializing the enveloping dense matrix at runtime.

```
%m = arith.constant sparse<
  [ [0, 0], [0, 7], [1, 2], [4, 2], [5, 3], [6, 4], [6, 6], [9, 7]],
    [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]> : tensor<10x8xf64>
%s = sparse_tensor.convert %m : tensor<10x8xf64> to tensor<10x8xf64, #CSR>
```

Lastly, the SparseTensor dialect provides operations that facilitate progressively lowering, first from annotated kernels to an intermediate form that only stores and iterates over nonzero elements, without fully committing to an underlying storage format yet, then from this intermediate form to a bufferized form with actual sparse storage schemes, and then finally to executable code that runs on the target hardware. In the intermediate form, it is useful to have some primitives that provide sparsity-specific functionality while still hiding full implementation details of the underlying sparse storage formats. For instance, the indices operation returns the indices array of the sparse storage format at the given dimension for the given sparse tensor through a one-dimensional array, called a memref after bufferization.

An example of the indices operation is shown below:

```
%ind = sparse_tensor.indices %t, %c3 : tensor<10x10x10x10xf64, #SparseTensor> to memref<?xindex>
```

Other operations related to sparsity consist of obtaining the pointers or values array and inserting elements into a sparse tensor, possibly assisted by expanding and compressing access patterns within innermost loops, as further explained later in this section.

4.3 Sparse Tensor Transformations

The sparse compiler itself is implemented as a set of lowering transformations that reside in the `SparseTensor` dialect. These transformations take care of lowering annotated kernels, i.e., operations on sparse tensor types, to imperative constructs and sparse storage formats that only store and iterate over the nonzero elements, possibly optimized with parallelization or vectorization of the generated loops by means of parallel and SIMD operations in the `SCF` and `Vector` dialects. In true MLIR fashion, transformation definitions separate **mechanism**, i.e., how to modify or lower an intermediate form, from **policy**, i.e., which transformations should be applied and in what order to get the best-performing result.

The MLIR sparse compiler transformations closely follow the *sparse iteration model* of TACO [48]. Starting with a tensor index notation expressed in the `Linalg` dialect, a topologically sorted iteration graph is constructed that reflects the required order on the implicit indices with respect to the dimension ordering of each sparse tensor. Next, iteration lattices are constructed for the tensor expression for every index in topological order. Each iteration lattice point consists of a conjunction of tensor indices together with a tensor (sub)expression that drive the actual sparse code generation, consisting of a straightforward mapping to loops and conditionals using the `SCF` dialect. The resulting sparse intermediate form that only (co-)iterates over the nonzero elements is subsequently bufferized to another intermediate form with actual sparse storage schemes, and then finally lowered to executable code that runs on the target hardware, typically by handing off LLVM IR to the LLVM back-end compiler.

As a simple example, consider the following straightforward `Linalg` operation that scales the elements in a vector by a constant in-place, viz. `x[i] *= c`. This particular operation takes the vector as parameter in the output clause but defines the body in terms of the individually "yielded" scalar multiplications. As for any `Linalg` operation, the actual decomposition into a loop is implicit.

```
%0 = linalg.generic #trait_scale
  outs(%vecx: tensor<?xf64, #SparseVector>) {
  ^bb(%x: f64):
    %1 = arith.mulf %x, %c : f64
    linalg.yield %1 : f64
  }
  -> tensor<?xf64, #SparseVector>
```

The steps discussed above reveal that the loop-body only needs to execute for nonzero elements in the sparse vector. This yields the following intermediate form that updates these elements in place using primitives of the `SparseTensor` and `SCF` dialects.

```
(1)  %x_pointers = sparse_tensor.pointers %vecx, %c0
(2)  %x_values = sparse_tensor.values %vecx
(3)  %x_lo = memref.load %x_pointers[%c0] : memref<?xindex>
(4)  %x_hi = memref.load %x_pointers[%c1] : memref<?xindex>
(5)  scf.for %ii = %x_lo to %x_hi step %c1 {
(6)    %0 = memref.load %x_values[%ii] : memref<?xf64>
(7)    %1 = arith.mulf %0, %c : f64
(8)    memref.store %1, %x_values[%ii] : memref<?xf64>
(9)  }
```

Lines (1–2) fetch the pointer and value buffers for the vector. Lines (3–4) initialize the loop bounds for the subsequent `for`-loop at lines (5–9). Within the body, line (6) fetches each explicitly stored value, multiplies it with the constant in line (7), and stores it back into the value buffer in line (8).

The sparse iteration model also provides an elegant way to generate code that implements *co-iteration*, i.e., iterating over a disjunction or a conjunction of several sparse data structures simultaneously. Consider, for example, computing the inner product `x += a[i] * b[i]` of two sparse vectors, represented by the following `Linalg` operation:

```
%0 = linalg.generic #trait_inner
  ins(%veca, %vecb: tensor<?xf64, #SparseVec>,
                    tensor<?xf64, #SparseVec>) outs(%acc: tensor<f64>) {
    ^bb(%a: f64, %b: f64, %x: f64):
      %0 = arith.mulf %a, %b : f64
      %1 = arith.addf %x, %0 : f64
      linalg.yield %1 : f64
  }
  -> tensor<f64>
```

Then the sparse iteration model reveals that the body only needs to execute when both elements of the sparse vectors are nonzero. This eventually yields in the following `while` construct of the `SCF` dialect, presented in a reduced and simplified form to enhance readability (mainly by omitting some machinery needed to represent reduction and induction cycles in SSA form).

```
(01)  %a_lo = memref.load %a_pointers[%c0] : memref<?xindex>
(02)  %a_hi = memref.load %a_pointers[%c1] : memref<?xindex>
(03)  %b_lo = memref.load %b_pointers[%c0] : memref<?xindex>
(04)  %b_hi = memref.load %b_pointers[%c1] : memref<?xindex>
(05)  %result = scf.while (...) {
(06)    %cond = %ia < %a_hi && %ib < %a_hi
(07)    scf.condition(%cond)
(08)  } do {
(09)    ...
(10)    %ixa = memref.load %a_indices[%ia] : memref<?xindex>
(11)    %ixb = memref.load %b_indices[%ib] : memref<?xindex>
(12)    %i = min(%ixa, %ixb)
(13)    %acc = scf.if (%ixa == %i && %ixb == %i) -> (f64) {
(14)      %0 = memref.load %a_values[%ia] : memref<?xf64>
(15)      %1 = memref.load %b_values[%ib] : memref<?xf64>
(16)      %2 = arith.mulf %0, %1 : f64
(17)      %3 = arith.addf %acc_in, %2 : f64
(18)      scf.yield %3 : f64
(19)    } else {
(20)      scf.yield %acc_in : f64
(21)    }
(22)    update %ia++ if %ixa == %i
(23)    update %ib++ if %ixb == %i
(24)  }
```

Here, lines (1–4) set up the loop bounds from already fetched pointer buffers for the subsequent `while`-loop at lines (5–24). The conditional clause at lines (5–8) iterates as long as neither elements in `a` or `b` are exhausted. Within the body at lines (8–24), the minimum of the indices of these elements is determined at line (12). If both indices are equal, the `if`-operation at lines (13–18) executes, which accumulates the products at line (17). The yields at line (18) and (20) bring the accumulation cycle back in SSA form. Finally, lines (22–23) advance the element of either vector or both, before iterating back into the loop. The loop exits when either vector exhausts its stored elements.

4.4 Sparse Tensor Transformations for Output

Sparse tensor outputs are handled with direct insertions in pure lexicographical index order if all loops that correspond to left-hand-side indices are outermost, since this allows simply appending new elements to the sparse storage formats without sorting or data movement. Otherwise, the sparse compiler uses a technique called access pattern expansion or workspace [7, 28, 35, 50, 68, 69], which is a well-known way of efficiently dealing with sparse insertions, as further explained using the following example of a `Linalg` representation of `C[i,j] = A[i,k] * B[k,j]`.

```
%C = linalg.matmul ins(%A,%B: tensor<?x?xf64,#CSR>,tensor<?x?xf64, #CSR>) -> tensor<?x?xf64,#CSR>
```

With row-wise storage for all matrices, the reduction loop cannot appear innermost, which prohibits simply inserting in pure lexicographical index order.

Consequently, the sparse compiler transformations lower this kernel to the following sparse intermediate form, again presented in a slightly reduced and simplified form to enhance readability.

```
(01)  scf.for %i = %c0 to %m step %c1 {
(02)    %c_i_values, %filled, %added, %count = sparse_tensor.expand %C
(03)    ...
(04)    scf.for %kk = %a_lo to %a_hi step %c1 {
(05)      %k = memref.load %a_indices[%kk] : memref<?xindex>
(06)      %aik = memref.load %a_values[%kk] : memref<?xf64>
(07)      ...
(08)      scf.for %jj = %b_lo to %b_hi step %c1 {
(09)        %j = memref.load %b_indices[%jj] : memref<?xindex>
(10)        %bkj = memref.load %b_values[%jj] : memref<?xf64>
(11)        %cij = memref.load %c_i_values[%j] : memref<?xf64>
(12)        %0 = arith.mulf %aik, %bkj : f64
(13)        %1 = arith.addf %cij, %0 : f64
(14)        scf.if (not %filled[%j]) {
(15)          memref.store %true, %filled[%j] : memref<?xi1>
(16)          record insertion as %added[ %count++ ] = %j
(17)        }
(18)        memref.store %1, %c_i_values[%j] : memref<?xf64>
(19)      }
(20)    }
(21)    sparse_tensor.compress %C, %c_i_values, %filled, %added, %count
(22)  }
```

The resulting code consists of a loop at line (1) that iterates over all rows of `A` and `C` in a dense manner, a loop at line (4) that iterates over stored elements in each row of `A`, and a loop at line (8) that iterates over stored elements in rows of `B`. The parts at lines (5–6) and (9–10) fetch values and indices as seen earlier for sparse tensor inputs.

Specific to the sparse tensor output is the `expand` operation before a next row of `C` at line (2), which yields two initially zeroed-out arrays `c_i_values` and `filled` with sizes that suffice for a dense innermost dimension (viz. a fully expanded row of `C`). Arrays `added` and scalar `count` are used to keep track of new indices when a false value is encountered in the `filled` array. The internal allocation and initialization are actually moved before the loop (or preferably even shared between loops of different kernels) by a later transformation, so that these dense assignments are amortized over as many iterations as possible. When a new value is detected at line (14) by encountering a false value at the corresponding position in `filled`, the insertion is recorded in the bookkeeping at lines (15–16). Reading from and writing to an expanded row in `C` is done at line (11) and line (18), respectively. When the row in `C` is exhausted, the `compress` operation at line (21) executes. This operation first sorts the indices of the new entries in the row, which enables subsequent insertion in pure lexicographical index order without costly data movement. Then the dense arrays are reset in a sparse fashion by only iterating over changed elements using the indices stored in `added`, which keeps the amount of work proportional to the number of stored elements rather than the full dimension size.

5 SPARSE COMPILER USAGE

The previous section gave details on the design and implementation of sparse compiler support in the MLIR compiler infrastructure. This section explores two ways to use this new support: end-to-end support for array languages and sparse state space search for testing or library development.

5.1 End-to-end Sparse Compiler Support for Array Languages

Our long-term north star vision centered around sparse tensor types as first-class citizens enables researchers to independently start developing new front-ends for array languages that incorporate some form of sparsity annotations as well as adding new MLIR transformations that implement alternative and improved sparse compiler pipelines. This vision would result in the retargetable approach illustrated in Figure 3, where several front-ends and several sparse compilers share the MLIR infrastructure.

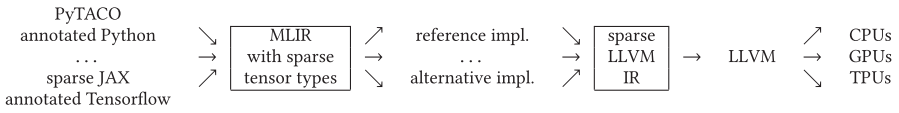

Fig. 3. Overview of retargetable sparse compiler support. Multiple front-ends map different sparse array languages, such as PyTACO, to a shared high-level intermediate representation, such as `Linalg` with sparse tensor types. Alternative sparse compiler pipelines, like the reference implementation, lower this intermediate to a form that only stores and iterates over nonzero elements. This intermediate is handed off to a retargetable back-end compiler, like LLVM.

All paths eventually generate sparse code in some intermediate form, such as LLVM IR, which can be handed off to a back-end compiler, like LLVM, to generate code for various target architectures.

The end product enables nonexpert programmers to exploit sparsity easily, since, by merely adding annotations, a single sparsity-agnostic program in, e.g., Python could map to a wide range of sparse implementations, each tailored to specific instances of the same problem, in terms of both sparsity properties and characteristics of the target architecture. MLIR ships with an initial reference implementation of the sparse compiler transformations. In addition, as an example of end-to-end support for an array language, we added PyTACO [49], which is TACO's Python-based domain-specific language for expressing sparse tensor algebra computations.

For example, the following PyTACO code expresses an element-wise matrix addition. The first three lines define tensors `A`, `B`, and `C`. The fourth line defines the two index variables `i` and `j`. The last line describes the actual computation, using a tensor index notation to describe how each element in the result tensor can be computed from elements in the two source tensors.

```
A = tensor([1024, 1024], [compressed, dense])
B = tensor([1024, 1024], [compressed, dense])
C = tensor([1024, 1024], [compressed, dense])
i, j = get_index_var(2)
C[i,j] = A[i,j] + B[i,j]
```

PyTACO supports implicit broadcast and implicit reduction. If an index variable is defined in the iteration space but does not appear in the expression of a tensor operand, the tensor operand is broadcast along the dimension represented by the index variable. If an index variable appears in the expression of some tensor operands but not in the expression of the destination tensor, then the corresponding dimension is reduced on the smallest sub-expression that captures the use of the index variable. The following code illustrates these two rules:

```
C[i,j] = A[i,j] + B[i]          =>  C[i,j] = A[i,j] + broadcast(j, B[i])
D[i] = A[i,j] + B[i,j] + C[i]   =>  D[i] = sum(j, A[i,j] + B[i,j]) + C[i]
```

To support PyTACO, we implemented Python classes for sparsity annotations, index variables, tensors, tensor accesses, and tensor expressions, as well as the dunder method `__getitem__` to process tensor accesses. A tensor access is a leaf node. Tensor expressions can participate in operations, such as additions and multiplications, to construct more complicated expressions. We also implemented the dunder method `__setitem__` to assign a tensor expression to the left-hand-side tensor. When a tensor with an assignment is evaluated, we generate the MLIR representation for the tensor assignment. The `Linalg` dialect has a generic operation that can be used to express the PyTACO tensor computation with only one difference. That is, a reduction in the generic operation is performed on the whole expression, not on the smallest expression that captures the use of the index variables as in the PyTACO. As such, to implement the PyTACO semantics correctly, we need to break the following PyTACO expression:

```
D[i] = A[i,j] + B[i,j] + C[i]
```

into the following two generic operations:

```
T[i] = A[i,j] + B[i,j]
D[i] = T[i] + C[i]
```

When we need to introduce such temporary tensors, we use heuristics to estimate the sparsity for each dimension in the temporary tensor. In particular, if an operation only preserves zero values from both source operands, such as an addition, we choose `compressed` format for a destination dimension only if both source dimensions are `compressed`. If an operation preserves zero values from either source operand, such as a multiplication, we choose `compressed` format for a destination dimension if at least one of its source dimensions is `compressed`. After generating the MLIR code for the tensor assignment, we invoke MLIR compilation passes followed by the MLIR JIT execution engine to execute the resulting runnable and retrieve the result for the tensor.

5.2 Sparse State Space Search

The MLIR infrastructure provides a Python interface for building, compiling, and running MLIR IR from code. This interface provides a convenient way to loop over all possible sparse storage formats and compiler optimizations for a particular kernel. Given the matrix multiplication kernel in which only one matrix is sparse, for example, the code below can be used to exhaustively explore all possible sparsity attributes `attr` and compiler strategies `opt` for building and running the kernel:

```
for level in [ [dense,      dense], [dense,      compressed],
               [compressed, dense], [compressed, compressed] ]:
  for ordering in [ ir.AffineMap.get_permutation([0, 1]),
                    ir.AffineMap.get_permutation([1, 0]) ]:
    for ptrWidth in [0, 8, 16, 32, 64]:
      for indxWidth in [0, 8, 16, 32, 64]:
        attr = st.EncodingAttr.get(level, ordering, ptrWidth, idxWidth)
        mlir = buildSpMM(attr)
        for opt in compilerStrategies:
          exec = compileKernel(mlir, opt)
          result = runKernel(exec, input)
          verify(result)
```

This approach is useful to stress test the actual sparse compiler implementation, since the computed result should be independent of the actual sparsity annotations or compiler optimizations used (at least within acceptable numerical differences due to floating-point reassociation). However, the same approach can also be used by an expert programmer to exhaustively explore the performance of a library method within the full state space of possible sparse implementations and optimizations before deciding on which one to ship into production.

The loop nest above shows that choosing a proper format for a *single* sparse matrix already gives rise to a state space of 200 configurations. For kernels with higher-dimensional or multiple sparse tensors, the size of the state space grows further. Then, when combined with different compiler optimization strategies, different target architectures, and different characteristics of the sparse tensor inputs, the combinatorial explosion of the state space becomes apparent.

Although long offline running times are acceptable while exhaustively searching for the best possible implementation of a library method that will be widely used, at some point exploring the full state space may become infeasible, and the programmer may have to resort to using machine learning for this exploration [77].

6 EXPERIMENTAL RESULTS

Even though our long-term objective is to provide an excellent, reusable sparse ecosystem that simplifies sparse code generation for a wide variety of target architectures (CPUs, GPUs, and TPUs), MLIR ships with a first reference implementation that provides a fully functional implementation of the sparse ecosystem (for CPUs only at the moment). In this section, we present an experimental validation of this initial reference implementation compared to the state-of-the-art TACO compiler [49]. All measurements were taken on an Intel Xeon W2135 3.7GHz.

6.1 Sparse Tensor Input

To validate the quality of reading tensors from files, we measured the time taken to read and pack tensors to an all-dimensions `compressed` format for a few tensors with varying dimensions from the Matrix Market [12] and FROSTT [78]. For TACO, we used the built-in timing benchmark, which invokes GCC during execution to compile generated code. For fair comparison, we compensated the reports to exclude compilation time. For MLIR, we used identical timing methods but the LLVM JIT compiler as back-end. Also, since MLIR reads tensors from an extended FROSTT format with extra size metadata in the header, it has a slight advantage reserving memory prior to reading. Table 1 shows the results, with the relative speedup of MLIR over TACO for the total reading and packing time in the last column. The latter gives clear evidence that the reference implementation provides state-of-the-art performance for reading sparse tensors.

Table 1. Sparse Tensor Input (time in ms)

			MLIR		TACO		
Tensor	**Dim**	**nnz**	**Read**	**Pack**	**Read**	**Pack**	**Speedup**
fidap011	2	1,091,362	228	136	262	142	1.11x
nell-2	3	76,879,419	9,568	2,055	13,303	6,974	1.74x
uber	4	3,309,490	418	107	594	398	1.89x
vast-2015	5	26,021,945	3,796	1,728	5,191	2,832	1.45x

6.2 Sparse Linear Algebra with Sparse Output

To validate the quality of the generated code with a sparse output, we measured the runtime of the SpMSpM kernel `C[i,j] = A[i,k] * B[k,j]` that was discussed in Section 4.4, where the output is a sparse matrix with a nonzero structure that depends on the nonzero structures of the input sparse matrices. Table 2 summarizes results when applying the kernel generated by both MLIR and TACO (explicitly combining the assemble-compute steps for the latter, since these are separated by default) to various $n \times n$ uniform random sparse matrices with a fixed density of $\rho_{A,B} = 0.01$, illustrated on the left in Figure 4, as well as to a few square matrices from the Matrix Market [12] and SuiteSparse [24, 25] used for both input arguments.

The table shows the resulting density ρ_C for the output sparse matrix, measured runtimes, and relative performance improvement of MLIR over TACO. Since both sparse compilers generate more or less the same code, as expected, performance is similar. Note that the kernel's performance depends on the way heap memory for the output matrix data structure is reserved and reallocated dynamically during insertions, since this is a big part of the execution profile.

Table 2. SpMSpM Kernel with Sparse Matrix Output (time in ms)

Matrix	n	nnz_A	ρ_C	**MLIR**	**TACO**	**Speedup**
random1K	1,024	10,458	0.10	3.4	3.8	1.12x
random2K	2,048	42,321	0.19	23.9	27,1	1.13x
random4K	4,096	167,628	0.34	211.5	225.3	1.07x
random8K	8,192	671,558	0.56	1,1650.3	1,731.1	1.05x
af23560	23,560	484,256	0.0044	44.3	52.0	1.19x
fidap011	16,614	1,091,362	0.0164	142.1	151.8	1.07x
xenon2	157,464	3,866,688	0.0006	348.0	371.0	1.07x
cage13	445,315	7,479,343	0.0003	1,607.8	1670.1	1.04x

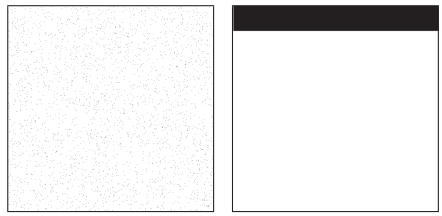

Fig. 4. Sparse matrices (uniform random and row band).

6.3 Sparse Linear Algebra with SIMD Optimizations

To demonstrate the impact of sparse storage format selection and compiler optimizations on performance, consider the following sparse matrix times vector operation, where we have a choice for attribute `SpMat`:

```
%x = linalg.matvec ins(%A, %b: tensor<?x?xf64, #SpMat>, tensor<?xf64>)-> tensor<?xf64>
```

Obvious choices for an unstructured sparse matrix `A` as input would be formats like CSR or DCSR, with the latter favoring sparse matrices that have many empty rows. However, for row band matrices, illustrated on the right in Figure 4, where many rows are empty but non-empty rows are dense, using an encoding with `compressed` for the outermost dimension and `dense` for the innermost dimension would make more sense. Let's call this format CDR. This format not only enables the compiler to generate an efficient storage scheme that matches the input data but also composes well with other compiler optimizations, such as vectorization. Eventually, the sparse code generated for CDR by MLIR looks as follows, where the `Vector` dialect is used to express a SIMD reduction in an architectural-neutral manner in the loop at lines (3–8). The reduction operation at line (9) sums the partial results up before storing the result into vector `x` at line (10).

```
(01)  scf.for %ii = %a_lo to %a_hi step %c1 {
(02)    ...
(03)    %acc = scf.for %j = %c0 to %n step %c16 {
(04)       ...
(05)       %mul = arith.mulf %aij, %bj : vector<16xf64>
(06)       %acc_out = arith.addf %acc_in, %mul : vector<16xf64>
(07)       ...
(08)    }
(09)    %xi = vector.reduction <add>, %acc : vector<16xf64> into f64
(10)    memref.store %xi, %x[%i] : memref<?xf64>}
(11)  }
```

Table 3 compares the performance of the code generated for CSR, DCSR, and CDR when applied to various $n \times n$ row band matrices where the first 1,000 rows are kept dense, so that density ρ_A decreases as size n increases. To take full advantage of AVX512 instructions on the target architecture along dense rows, vectorization was enabled for both compilers (built-in for MLIR and by passing extra native flags to the TACO compilation).

Table 3. SpMV Kernel with SIMD Optimization (time in ms)

			MLIR			TACO			
Matrix	n	ρ_A	**CSR**	**DCSR**	**CDR**	**CSR**	**DCSR**	**CDR**	**Speedup**
rowband8K	8,192	0.12	7.88	7.85	5.22	10.99	10.78	6.62	1.27x
rowband16K	16,384	0.06	15.76	15.69	10.52	19.77	19.21	11.90	1.13x
rowband32K	32,768	0.03	31.51	31.41	21.07	38.63	37.27	24.79	1.18x
rowband64K	65,536	0.02	77.28	72.19	42.18	77.55	76.84	47.07	1.12x

As expected, CSR and DCSR perform similarly, with a slight advantage in the doubly compressed format due to skipping empty rows completely. CDR performs best. In addition, MLIR takes full advantage of SIMD, demonstrating the utility of integrating sparse compilation into a modern multi-level compiler infrastructure.

6.4 Sparse Linear Algebra and State Space Search

To illustrate the use of sparse state space search to find a suitable sparse storage scheme, we conducted experiments with the well-known **sampled dense-dense matrix multiplication (SDDMM)** kernel [64] that samples the result of dense matrix multiplication by means of element-wise multiplication with a sparse matrix, expressed as `X[i,j] = S[i,j] * A[i,k] * B[k,j]` in tensor index notation for a sparse matrix `S`.

Figure 5 plots the runtime of this kernel for different versions generated by MLIR using various sparse storage schemes for `S` (left-to-right) and compiler optimization strategies (sorted front to back). The figure clearly visualizes how an expert programmer can find the best storage format and optimization strategy by looking for the minimum runtime in the plot (lowest point), here corresponding to DCSR and a particular optimization setting.

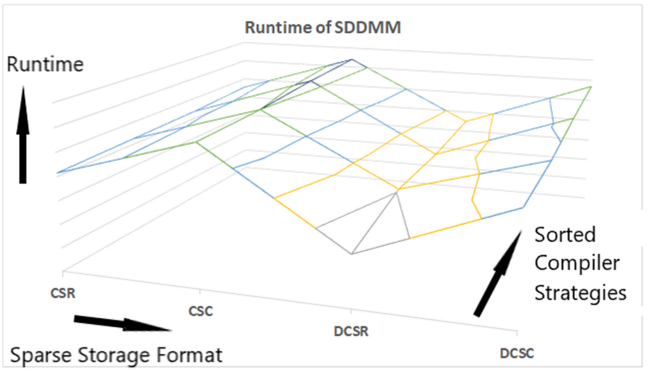

Fig. 5. Sparse runtime state space for SDDMM (lower is better).

6.5 Sparse Tensor Algebra on 3-Dimensional Tensors

To validate the quality of the generated code for higher-dimensional tensor algebra, we measured the runtime of the Matricized Tensor Times Khatri-Rao Product, or MTTKRP for short. This kernel, expressed in tensor index notation as `A[i,j] = B[i,k,l] * D[l,j] * C[k,j]` (viz. two reduction loops) for a sparse 3-dimensional tensor B, forms the bottleneck of various problems in data analytics. Table 4 shows the runtime of this kernel for various tensors of FROSTT [78] and HaTen2 [43]. Following the TACO convention, the unknown output dimension is set to 42. For example, the $12092 \times 9184 \times 28818$ input sparse tensor `nell-2` is assumed to generate a 12092×42 output matrix. For both MLIR and TACO, we measured the time for the sparse versions generated for formats with an outermost `dense` and two innermost `compressed` dimensions and default lexicographical index order (DSS-ijk), as well as an alternative format with dimension ordering that interchanges the two innermost dimension (DSS-ikj). The latter may come at the expense of slightly higher packing times to sort from an original lexicographical order in the external format. The table reveals potential advantages of alternative dimension orderings.

Table 4. Matricized Tensor Times Khatri-Rao Product Kernel (time in ms)

		MLIR		TACO		
Tensor	**nnz$_B$**	**DSS-ijk**	**DSS-ikj**	**DSS-ijk**	**DSS-ikj**	**Speedup**
darpa1998	28,436,033	1,548	1,556	1,655	1,801	1.07x
nell-2	76,879,419	1,752	1,351	2,095	1,668	1.24x
freebase-sampled	139,920,771	8,504	8,379	12,168	12,531	1.45x
nell-1	143,599,552	12,063	11,878	13,538	12,909	1.09x

6.6 Sparse Tensor Algebra with Sparse Output

To validate the quality of code generated for tensor algebra with a sparse output, we measured the runtime of the TTM kernel `C[i,j,k] = A[i,j,l] * B[k,l]` for sparse tensors `A` and `C` and a dense matrix `B`. Using formats with an outermost `dense` and two innermost `compressed` dimensions together with default index storage allows both MLIR and TACO to generate direct insertions in pure lexicographical index order, without the need for access pattern expansion (a.k.a. workspace). Table 5 summarizes results when applying the kernel generated by both MLIR and TACO, again combining the assemble-compute steps for the latter and using 42 for the now unknown input dimension. The table shows the number of nonzero elements in the input and output tensors as well as the runtime for the code generated by MLIR and TACO. Even though the generated code is similar, TACO inlines the lexicographical insertion code, whereas MLIR uses individual virtual method calls into a support library, which is something we plan to improve in the future. As before, performance depends on the way heap memory for the output tensor is reserved and reallocated dynamically during the insertions.

Table 5. TTM Kernel with Sparse Tensor Output (time in ms)

Tensor	**nnz$_A$**	**nnz$_C$**	**MLIR**	**TACO**	**Speedup**
darpa1998	28,436,033	3,209,808	6,427	6,478	1.01x
nell-2	76,879,419	14,169,330	11,382	11,574.1	1.02x
nell-1	143,599,552	729,641,514	103,793	137,468.0	1.32x

7 RELATED WORK

This section discusses related work, organized by sparsity support in compilers and libraries.

7.1 Sparse Compilers

To the best of our knowledge, Bik and Wijshoff were the first to propose the concept of treating sparsity merely as a property and letting a sparse compiler generate sparse code automatically from a sparsity-agnostic definition of the computation [9], which resulted in the *MT1 sparse compiler* for linear algebra [7, 10]. This Fortran compiler first evaluates an attribute grammar to associate sparse conditions for execution with all expressions and statements in the input program. Then, driven by annotations and automatic nonzero structure analysis of representative sparse input matrices [11], the program is automatically converted into a form that only stores and operates on nonzero elements for unstructured and structured sparse matrices. Code generation featured conjunctions and disjunctions, access pattern expansion, and sparse matrix outputs. Using this sparse compiler for automatic sparse library generation was demonstrated in [8] for linear algebra primitives.

The concept of generating sparse code from dense abstractions is shared with subsequent work. *The Vienna-Fortran/HPF sparse extensions* [88] provide High-Performance Fortran language features where users can characterize a matrix as sparse and specify the associated representation for its underlying compiler. *The Bernoulli project* [53, 61] generates efficient serial and distributed sparse matrix code from HPF dense DO-ANY loops through relational algebra. By viewing arrays as relations and execution of loop nests as evaluation of relational queries, the problem of effectively traversing the sparse iteration space becomes finding optimal select and join schedules. However, users must provide access methods for their sparse data storage, which can be a significant learning curve. The *SIPR framework* [69] applies compilation to sparse algorithms with auxiliary data structures, such as a sparse accumulator [33]. It represents sparse programs as a combination of dense algorithms, utilizing existing C++ libraries, and dynamic sparse storage specifications, mapping array references and loop iterations to storage formats and array access descriptors. SIPR supports a limited number of sparse matrix formats (Compressed Stripe Storage).

Compared to the MT1 compiler and the follow-on systems above, the system described in this article produces code for tensor algebra instead of linear algebra. It also generates code from high-level tensor expressions instead of Fortran code with arrays. Furthermore, it supports arbitrary fusion of sparse and dense tensor algebra expressions, leading to code that co-iterates over any number of matrices. Finally, it generates code using a different compilation approach.

Sparse compilation was formalized and generalized to sparse tensor algebra in *TACO (Tensor Algebra Compiler)* [48–51]. Starting with a "loopless" tensor index notation, the programmer annotates tensors as sparse, and the framework generates corresponding C++ or CUDA code. TACO was the first to propose the elegant way of expressing desired sparse storage schemes with a combination of per-dimension level types and a dimension ordering. The dimension level types were expanded by Chou et al. [18] to include more sparse storage formats. Kjolstad et al. [50] were also the first to present a sparse iteration model that drives sparse code generation by topologically sorting the iteration graph to find a suitable index order and using iteration lattices to emit loops, conditions, and tensor expressions for every index in topological order. For this, TACO provides the co-iteration formulation that can be used to generate code to co-iterate over any number of sparse and dense tensors, which is necessary for general kernel fusion. Kjolstad et al. [50] and Senanayake et al. [76] also extended the TACO compiler with a scheduling language that lets users (or automatic systems) organize the iteration over tensor expressions, which lets them tile, control fusion/fission, statically load-balance, and generate GPU code for sparse tensor algebra kernels. Sparse tensor support in MLIR borrows heavily from the foundation laid by TACO, with additional challenges of making our IR fit well within the MLIR stack, automatically benefiting from optimizations from upper layers and vectorization from lower layers, and generalizing to sufficient support of diverse hardware in underlying layers.

Taichi [41] is a programming language that offers a data structure-agnostic interface for writing computation code. The user specifies the data structure with different sparsity properties independent of the code to create a wide range of sparse data structures. As with the approaches above, decoupling data structures from computation simplifies experimenting with different data structures without the need to change the actual computation code. The system described in this article, however, supports arbitrary tensor algebra expressions with multiple sparse operands.

COMET [62, 86] is an MLIR-based compiler infrastructure for dense and sparse tensor algebra computations. It uses a dimension-wise sparse storage scheme and a code generation algorithm similar to TACO, but has more performance portability building upon MLIR. Users describe tensor algebra expressions with Einstein summation notation and sparse tensor format tags in COMET's DSL. The DSL is mapped to a custom Sparse Tensor Algebra dialect, which is progressively lowered to LLVM IR through MLIR dialects. Among compiler-based approaches, COMET's is the closest

to ours. COMET has more optimizations, such as data reordering [58], while our work is focused on enabling native and seamless sparse tensor support throughout the MLIR stack. This lets us leverage existing dense MLIR code with just minor changes, including integrations with higher-level frameworks like TensorFlow [1] and JAX [13]. We believe both COMET and our work can benefit from each other and look forward to when COMET is open source.

Henry et al. [40] generalized the sparse tensor algebra compilation theory from TACO [51] to support compilation of general dense and sparse array expressions, laying the foundation for a sparse NumPy-style system. In this system, arrays (tensors) can have any implicit fill value and any function can be computed across sparse and dense arrays. The new scheme covers sparse iteration spaces outside of union (addition) and intersection (multiplication) for arbitrary user-defined functions. It can also iterate over slices or strides of sparse arrays, allowing for a much wider range of kernels and applications. Our infrastructure can be extended to support this concept.

Another line of work is to directly optimize sparse code, i.e., codes that deal with compressed data structures. These optimizations are applicable to our work. The main difference is that users have to write the starter code, which can be tedious and error-prone, especially when processing high-dimensional tensors. *SPARSITY* [42] provides a framework that automatically applies register and cache optimizations to sparse matrix codes based on machine-specific performance models. ***LL (Little Language)*** [4] is a small functional language aiming to increase programming productivity of sparse operations. It provides built-in nested list and pair types, with which users can naturally represent compressed matrices in many formats. Sparse computations are defined through a data flow graph. The compiler can generate efficient, fully verified C code. *Sympiler* [17] symbolically analyzes sparse codes at compile time and performs code optimizations specific to sparsity patterns.

The ***Sparse Polyhedral Framework (SPF)*** [83] provides code and data transformations for non-affine loop bounds and array index expressions [91] that can be combined with existing transformation in a polyhedral framework. Each transformation controls whether a dimension in the sparse iteration space is traversed fully (*make-dense*) or sparsely (*compact*), with an option to pad the underlying data storage for that dimension (*compact-and-pad*). The generated code is akin to converting sparse tensors to specific layouts and then performing the computation. SPF can cover a large variety of tensor storage formats, at the expense of users having to write a sequence of transformations to produce the desired computations. *TIRAMISU* [5] is a polyhedral compiler for deep learning that maps sparse and recurrent neural networks to the polyhedral model in order to generate highly optimized sparse code for the target architecture. As opposed to these systems, the system described in this article can handle general tensor algebra expressions and can fuse across any number of sparse operations.

7.2 Sparse Libraries and Kernels

A large number of applications build upon sparse computation library primitives optimized for their target architectures instead of compilers. This section discusses how sparse libraries evolve over time to support increasing kernel requirements such as higher dimensionality, better programmability, and more data types, operations, and devices.

Classic sparse linear algebra binary libraries like *MKL* [93] and *cuSPARSE* [63] implement sparse basic linear algebra subroutines with few data types. More recent generic C++ libraries such as *Eigen* [34] and *CUSP* [23] allow writing math-like expressions and can support more data types. The *OSKI* [92] library offers automatically tuned sparse matrix CPU primitives for linear solvers with an easy-to-use interface. *PETSc* [6] is a library of scalable PDE solvers that has modular structure and multiple abstraction layers, making it easy to add support for new devices, kernels, data

types, and so forth, including new sparse matrix formats [54]. It can also be integrated with a compiler-based autotuning approach to further optimize its hotspot kernels [70].

Matrix algebra can be beneficial in graph processing [36]. The *GraphBLAS* [46] standard specifies a core set of general sparse matrix-based graph operations, e.g., generalized matrix operations over arbitrary semi-rings (replacing element-wise addition and multiplication with other operators), providing additional operation coverage to common sparse linear algebra libraries. There are many libraries implementing this standard [3, 16, 24, 95], targeting CPU and GPU devices.

In scientific computing, most common sparse tensor computations are tensor decompositions [52] and contractions [45, 66]. The *Cyclops Tensor Framework* [81, 82] is a C++ template library for distributed dense and sparse tensor algebra targeting the quantum chemistry domain. Through operator overloading, users can write tensor algebra expressions with Einstein summation notation directly in C++, e.g., `C["hij"] += A["ijhk"] * B["hkij"]`. It supports templated data types and can substitute element-wise operations in tensor addition and contraction with other operations through user-defined algebraic structures (e.g., semi-rings, groups, etc.). Among library-based approaches, CTF is the closest related work to ours due to its expressiveness. However, it supports few sparse storage formats, and we found that its sparse kernel performance lags behind a compiler-generated solution.

The rising popularity of deep learning poses new challenges to sparse libraries, e.g., new sparsity characteristics, kernels, data types, hardware accelerators, and so forth. While there are libraries such as Sputnik [31], cuSPARSELt [21], and LIBXSMM [38] adding new kernels and data types specific to deep learning, these kernels still have limited composability and portability.

Specializing sparse kernels for specific sparsity characteristics or hardware architecture can bring significant performance gains. Numerous applications implement custom sparse formats that better suit their sparsity patterns and target hardware [14, 15, 57, 65, 72, 74, 75, 80, 87].

Recent studies [67, 94, 96] on sparse matrix format classification using machine learning determine the best input storage format to maximize the performance of a particular kernel. Only a small number of matrix formats were compared, because each of them needs separate implementation. With our work, an arbitrarily large number of formats can simply be explored as compiler tuning knobs [77].

8 CONCLUSIONS AND FUTURE WORK

In this article, we proposed treating sparsity as a property, not a tedious implementation task, and showed how the concept of letting a sparse compiler generate sparse code automatically from a sparsity-agnostic definition of the computation is being integrated into the MLIR open-source compiler infrastructure. MLIR's ease of defining multi-level abstractions and transformations composes well with this idea of making sparsity a property. We discussed the advantages of sparse compilation as well as our north star vision of providing an excellent, reusable sparse ecosystem by introducing sparse tensor types as first-class citizens into MLIR. A reference implementation provides a fully functional implementation of this sparse ecosystem. We discussed two ways of using sparse compiler support in MLIR, namely, end-to-end support for sparse array languages and using a state space search for sparse library development. Lastly, we provided experimental validation that the reference implementation compares well to the state of the art in sparse compilation.

The MLIR project was in part started to provide an extensible and powerful compiler infrastructure for domain-specific languages. Of particular interest was dense tensor algebra for deep neural networks. But while dense tensor computations remain the bulk of machine learning workloads, sparse neural networks are increasingly being explored, whether weight-sparse, activation-sparse,

or input-sparse such as graph convolutional neural networks. With the MLIR sparse compiler, we seek to put compilation for sparse tensor algebra on the same strong footing as compilation for dense tensor algebra. We believe this is necessary to enable both efficient exploration and efficient computation of sparse computations in machine learning and in data analytics. In fact, classical compiler transformations for dense tensor computations, such as loop reordering, fusion, and fission, can be even more important for sparse computation, as getting loop ordering and fusion/fission wrong often leads to *asymptotically worse complexity*. This effect occurs even in staple computations like sparse matrix-matrix multiplication and sampled dense-dense matrix multiplication, which are used in sparse neural networks. The MLIR sparse compiler is positioned to address these challenges and we are therefore excited about its potential.

The contribution of this work has been the introduction of a reusable sparse ecosystem in a powerful and extensible open-source compiler infrastructure. By providing a solid foundation centered around sparse tensor types as first-class citizens together with a completely functional sparse compiler pipeline, future research can immediately explore new ideas and experiment with alternative approaches. Since transformations in MLIR compose well, experimental results will not be hampered by the lack of supporting optimizations, as is often the case with ad hoc prototypes of new ideas. Going forward, our hope is that the open-source nature of this project will not only benefit researchers that are new to the sparse domain but also solicit useful contributions from experts in the open-source community at large. We are looking forward to such contributions and hope that our project contributes to advancing sparse compiler research.

Although the system we have described in this article lays a strong foundation for sparse tensor algebra compilation in the MLIR infrastructure, there are several limitations that should be addressed in future work. First, the system does not currently support loop collapsing and tiling, which precludes certain locality-enhancing optimizations. Second, it does not currently support compilation to GPUs or other accelerators such as the Google TPU. Third, it supports formats that can be constructed from dense and compressed data structures, such as CSR, CSC, DCSR, DCSC, BCSR, and CSF, but not currently those that need more sophisticated data structures, such as ELL, DIA, HiCOO, ALTO, and hash maps, which can sometimes better take advantage of certain tensor structures or hardware features. And finally, it does not currently support automatic format selection. We believe all of these limitations can be addressed in future work. For example, tiling and mapping to GPUs could be expressed by adding a scheduling language, and more data structures could be supported by better format abstractions. To address these limitations, we invite contributions from the open-source and research communities.

Furthermore, we believe the performance of the system described in this article could be improved. As we have shown, the current system has excellent performance. However, properly mapping parallel and SIMD constructs to the parallelism provided by the target architecture is also on ongoing topic of investigation. Finally, we believe it is possible to expand the language supported by the compiler beyond tensor algebra. One such extension would be to support general array language constructs in the same intermediate representation, such as the application of nonlinear functions across tensors. A second extension would be to support kernels with complex access patterns such as sparse convolutions or direct solvers that exhibit "fill-in" on sparse tensors during the computation. A third extension would be to support the data reorganization operations and tensor generators, such as reshaping and concatenating tensors and generating identity tensors, that are common in machine learning frameworks. Finally, another interesting direction of future work would be to introduce set-oriented looping constructs to MLIR, which would enable a more gradual progressive lowering into sparse code, since the current direct lowering to `SCF` bridges a rather wide semantic gap. As before, we welcome the open-source and research communities to contribute to these and other extensions.

ACKNOWLEDGMENTS

The authors would like to thank the MLIR team, with a special shout-out to Mehdi Amini, Eugene Burmako, Diego Caballero, Albert Cohen, Stephan Herhut, Stella Laurenzo, Andrew Leaver, Jacques Pienaar, Thomas Raoux, River Riddle, Wren Romano, Sean Silva, Gus Smith, Matthias Springer, Reid Tatge, Jake van der Plas, Alex Zinenko, and Eugene Zhulenev.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for Large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[2] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 6 (1989), 73–95.

[3] Michael J. Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L. Willke, and Pradeep Dubey. 2016. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 313–322.

[4] Gilad Arnold. 2011. *Data-parallel Language for Correct and Efficient Sparse Matrix Codes*. University of California, Berkeley.

[5] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman Amarasinghe. 2020. TIRAMISU: A polyhedral compiler for dense and sparse deep learning. *arXiv preprint arXiv:2005.04091* (2020).

[6] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern Software Tools for Scientific Computing*. Springer, 163–202.

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

[8] Aart J. C. Bik, Peter J. H. Brinkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. 1998. The automatic generation of sparse primitives. *Transactions on Mathematical Software* 24 (1998), 190–225.

[9] Aart J. C. Bik and Harry A. G. Wijshoff. 1995. Advanced compiler optimizations for sparse computations. *J. Parallel and Distrib. Comput.* 31 (1995), 14–24.

[10] Aart J. C. Bik and Harry A. G. Wijshoff. 1996. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems* 7, 2 (1996), 109–126.

[11] Aart J. C. Bik and Harry A. G. Wijshoff. 1999. Automatic nonzero structure analysis. *SIAM J. Computing* 28, 5 (1999), 1576–1587.

[12] Ronald Boisvert, Roldan Pozo, and K. Remington. 1996. The Matrix Market Exchange Formats: Initial Design. NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD.

[13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: Composable transformations of Python+NumPy programs. http://github.com/google/jax.

[14] Aydın Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York, NY, 233. https://doi.org/10.1145/1583991.1584053

[15] Aydın Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536313

[16] Aydın Buluç and John R. Gilbert. 2011. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[17] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[18] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor Algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276493

[19] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. Association for Computing Machinery, New York, NY, 823–838. https://doi.org/10.1145/3385412.3385963

[20] Thomas F. Coleman. 1984. Large sparse numerical optimization. In *Lecture Notes in Computer Science, No. 165*, G. Goos and J. Hartmanis (Eds.). Springer-Verlag, Berlin.

[21] NVIDIA Corporation. 2021. cuSPARSELt: A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication. https://docs.nvidia.com/cuda/cusparselt/index.html.
[22] A. R. Curtis and J. K. Reid. 1971. The solution of large sparse unsymmetric systems of linear equations. *Journal Inst. Maths. Applics.* 8 (1971), 344–353.
[23] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusplibrary.github.io/. Version 0.5.0.
[24] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25.
[25] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1–25.
[26] Iain S. Duff. 1977. A survey of sparse matrix research. In *Proceedings of the IEEE.* 500–535.
[27] Iain S. Duff. 1985. Data structures, algorithms and software for sparse matrices. In *Sparsity and Its Applications*, David J. Evans (Ed.). Cambridge University Press, 1–29.
[28] Iain S. Duff, A. M. Erisman, and J. K. Reid. 1990. *Direct Methods for Sparse Matrices.* Oxford Science Publications, Oxford.
[29] Iain S. Duff, Roger G. Grimes, and John G. Lewis. 1989. Sparse matrix test problems. *ACM Trans. Math. Software* 15 (1989), 1–14.
[30] D. J. Evans. 1974. Iterative sparse matrix algorithms. In *Software for Numerical Mathematics*, D. J. Evans (Ed.). Academic Press, New York, NY, 49–83.
[31] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020.*
[32] Alan George and Joseph W. H. Liu. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Englewood Cliffs, NY.
[33] John Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in Matlab: Design and implementation. *SIAM J. on Matrix Analysis and Applications* 13, 1 (1992), 333–356.
[34] Gaël Guennebaud, Benoit Jacob, et al. 2010. Eigen. http://eigen.tuxfamily.org.
[35] Fred G. Gustavson. 1972. Some basic techniques for solving sparse systems of linear equations. In *Sparse Matrices and Their Applications*, Donald J. Rose and Ralph A. Willoughby (Eds.). Plenum Press, New York, NY, 41–52.
[36] F. Harary. 1969. *Graph Theory,* chs. 2, 13. Addison Wesley, Reading.
[37] Frank Harary. 1971. Sparse matrices and graph theory. In *Large Sparse Sets of Linear Equations*, J. K. Reid (Ed.). Academic Press, 139–150.
[38] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16).* IEEE, 981–991.
[39] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa M. Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive linearized storage of sparse tensors. *CoRR* abs/2102.10245 (2021). arXiv:2102.10245 https://arxiv.org/abs/2102.10245.
[40] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.
[41] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–16.
[42] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.
[43] Inah Jeon, Evangelos E. Papalexakis, Christos Faloutsos, Lee Sael, and U. Kang. 2016. Mining billion-scale tensors: Algorithms and discoveries. *International Journal on Very Large Data Bases (VLDB).* 25, 4 (2016), 519–544. DOI: 10.1007/s00778-016-0427-4
[44] Eun Jin Im and Katherine Yelick. 1998. Model-based memory hierarchy optimizations for sparse matrices. In *Workshop on Profile and Feedback-directed Compilation.*
[45] Daniel Kats and Frederick R. Manby. 2013. Sparse tensor framework for implementation of general local correlation methods. *Journal of Chemical Physics* 138, 14 (2013), 144101.
[46] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science* 51 (2015), 2453–2462.
[47] David R. Kincaid, John R. Respess, David M. Young, and Rober R. Grimes. 1982. Algorithm 586: ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Trans. Math. Softw.* 8, 3 (1982), 302–322.

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

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

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

[51] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor Algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[52] Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Review* 51, 3 (2009), 455–500.

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

[54] Pramod Kumbhar. 2011. *Performance of PETSc GPU Implementation with Sparse Matrix Storage Schemes*. Ph.D. Dissertation. Master's thesis, The University of Edinburgh (Aug. 2011).

[55] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'21)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

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

[57] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*. IEEE, 238–252.

[58] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. 2019. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*. 227–237.

[59] Joseph W. H. Liu. 1986. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Software* 12, 2 (1986), 127–148. https://doi.org/10.1145/6497.6499

[60] Ken J. Mann. 1982. Inversion of large sparse matrices: Direct methods. In *Numerical Solutions of Partial Differential Equations*, J. Noye (Ed.). North-Holland Publishing Company, Amsterdam, 313–366.

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

[62] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2021. COMET: A domain-specific compilation of high-performance computational chemistry. *arXiv preprint arXiv:2102.06827* (2021).

[63] Maxim Naumov, L. Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[64] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. 2018. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC'18)*. 32–41. https://doi.org/10.1109/HiPC.2018.00013

[65] Thomas C. Oppe and David R. Kincaid. 1987. The performance of ITPACK on vector computers for solving large sparse linear systems arising in sample oil reservoir simulation problems. *Communications in Applied Numerical Methods* 3, 1 (1987), 23–29.

[66] John A. Parkhill and Martin Head-Gordon. 2010. A sparse framework for the derivation and implementation of fermion algebra. *Molecular Physics* 108, 3–4 (2010), 513–522.

[67] Juan C. Pichel and Beatriz Pateiro-Lopez. 2019. Sparse matrix classification on imbalanced datasets using convolutional neural networks. *IEEE Access* 7 (2019), 82377–82389.

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

[69] William Pugh and Tatiana Shpeisman. 1998. SIPR: A new framework for generating efficient code for sparse matrix computations. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 213–229.

[70] Shreyas Ramalingam, Mary Hall, and Chun Chen. 2012. Improving high-performance sparse libraries using compiler-assisted specialization: A PETSc case study. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 487–496.

[71] J. K. Reid. 1974. Direct methods for sparse matrices. In *Software for Numerical Mathematics*, D. J. Evans (Ed.). Academic Press, New York, NY, 29–47.

[72] John R. Rice and Ronald F. Boisvert. 1985. *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag. 497 pages. https://doi.org/10.1007/978-1-4612-5018-0

[73] Youcef Saad. 1990. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. CSRD/RIACS.
[74] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM. https://doi.org/10.1137/1.9780898718003
[75] Nobuo Sato and W. F. Tinney. 1963. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Transactions on Power Apparatus and Systems* 82, 69 (1963), 944–950. https://doi.org/10.1109/TPAS.1963.291477
[76] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226
[77] Gus Henry Smith, Aart J. C. Bik, Penporn Koanantakool, and Phitchaya Mangpo Phothilimthana. 2022. ML-driven Auto-Configurator for Sparse Tensor Kernels in MLIR. Unpublished Manuscript.
[78] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. http://frostt.io/.
[79] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA^3'15)*. Association for Computing Machinery, New York, NY, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183
[80] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 1–7. https://doi.org/10.1145/2833179.2833183
[81] Edgar Solomonik and Torsten Hoefler. 2015. Sparse tensor algebra as a parallel programming model. *arXiv preprint arXiv:1512.00066* (2015).
[82] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 813–824.
[83] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.
[84] Parker Allen Tew. 2016. *An Investigation of Sparse Tensor Formats for Tensor Libraries*. M.Eng. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://groups.csail.mit.edu/commit/papers/2016/parker-thesis.pdf.
[85] Reginal P. Tewarson. 1973. *Sparse Matrices*. Academic Press, New York, NY.
[86] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A high performance sparse tensor Algebra compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC'21)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009
[87] William F. Tinney and John W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. In *Proceedings of the IEEE*. 1801–1809.
[88] Manuel Ujaldon, Emilio L. Zapata, Barbara M. Chapman, and Hans P. Zima. 1997. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems* 8, 10 (1997), 1068–1083.
[89] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730. http://arxiv.org/abs/1802.04730.
[90] M. Veldhorst. 1982. *An Analysis of Sparse Matrix Storage Schemes*. Ph.D. Dissertation. Mathematisch Centrum, Amsterdam.
[91] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices* 50, 6 (2015), 521–532.
[92] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (Jan. 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071
[93] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.
[94] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*. 94–105.
[95] Carl Yang, Aydin Buluc, and John D. Owens. 2019. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *arXiv preprint arXiv:1908.01407* (2019).
[96] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 94–108.
[97] Zahari Zlatev. 1991. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, Dordrecht.

Received February 2022; revised April 2022; accepted June 2022