A Mechanized Algebra of Verified Data Structures for Optimizing Sparse Tensor Programs

AMANDA LIU, Massachusetts Institute of Technology, USA
GILBERT LOUIS BERNSTEIN, University of Washington, USA
SHOAIB KAMIL, Adobe, USA
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA

In this paper, we introduce a verified framework for defining and composing sparse tensor formats. We extend the ATL tensor language and scheduling framework, which formerly could only express dense tensor kernels. We define a levelized abstraction to describe per-dimension tensor formats via their encoding routines, access and iteration functions, and formal properties enforcing soundness of the sparse structures as representations of the original dense tensors. Using this abstraction, we compositionally define format-agnostic, multidimensional compression and decompression functions that are used to express the top-level soundness theorem for these abstract sparse tensor formats. We then use this soundness theorem as an adjoint-pair rewrite theorem to introduce sparse data structures and iteration into a dense tensor kernel via the existing scheduling-rewrite framework of ATL. Overall, we are able to start with a program computing over dense operands and derive a proven semantically equivalent, optimized program computing over sparse structures. We further prove a minimal set of instances of the level-format abstraction, which can be composed and passed as parameters to compression to capture a broad range of canonical, multidimensional tensor-compression formats.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Formal software verification**; *Source code generation.*

Additional Key Words and Phrases: formal verification, sparse tensors, sparse formats, compilers, optimization

ACM Reference Format:
Amanda Liu, Gilbert Louis Bernstein, Shoaib Kamil, Adam Chlipala, and Jonathan Ragan-Kelley. 2026. A Mechanized Algebra of Verified Data Structures for Optimizing Sparse Tensor Programs. *Proc. ACM Program. Lang.* 10, PLDI, Article 183 (June 2026), 23 pages. https://doi.org/10.1145/3808261

1 Introduction

High-performance programs often involve efficiently computing pipelines of complex operations over multidimensional arrays, or tensors. It is common to store and compute over tensors in "dense" representations, with contiguous storage of cells. However, in many cases, the tensor operands are *sparse*, meaning that many of their values are zero. Exploiting sparsity allows computation to skip unnecessary operations and data movement by compressing zeros and maintaining index metadata that maps the compressed data back to its logical dense form. Different compression methods produce different metadata structures, giving rise to a variety of sparse formats.

Authors' Contact Information: Amanda Liu, Massachusetts Institute of Technology, Cambridge, USA, lamanda@mit.edu; Gilbert Louis Bernstein, University of Washington, Seattle, USA, gilbo@cs.washington.edu; Shoaib Kamil, Adobe, New York, USA, kamil@adobe.com; Adam Chlipala, Massachusetts Institute of Technology, Cambridge, USA, adamc@csail.mit.edu; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, USA, jrk@mit.edu.

ACM 2475-1421/2026/6-ART183
https://doi.org/10.1145/3808261

The choice of sparse format has major performance implications because the resulting metadata determines how values are accessed, iterated over, and laid out in memory. Some formats, such as HASH, optimize for fast random access using hash-indexed coordinates. Others, like compressed sparse row (CSR) and its higher-order variants, optimize for efficient iteration. Therefore, writing fast sparse tensor programs is complicated by format-specific logic.

This paper presents a verified approach to optimizing tensor kernels by introducing sparse tensor formats in a compositional, abstract way. We build upon the ATL scheduling framework, embedded in the Rocq proof assistant [Liu et al. 2022]. ATL was designed to optimize dense tensor programs by decoupling compute and storage order in a principled, algebraic manner. In this work, we conceptualize sparse formats as a kind of storage reordering within the ATL scheduling paradigm and extend the framework to support sparse format optimizations via the following contributions:

- A formal, level-based format abstraction capturing how a tensor is compressed on a per-dimension basis, including necessary properties for its soundness as a representation of the original dense tensor
- Multidimensional compress and decompress functions built from the level-based abstraction
- A format-agnostic soundness theorem of compress and decompress to be used as an ATL scheduling adjoint-pair rewrite
- Mechanized proofs of a core set of level-format instances that can be combined with ATL reshape operators to express many common multidimensional tensor-compression formats
- New core ATL language constructs for scatter and coiteration that allow iteration over compressed structures

Ultimately, we achieve an end-to-end verified, source-to-source optimization-derivation process. Starting from a simple, dense ATL program, we are able to derive an optimized ATL program operating over sparse structures while providing formal guarantees of the preservation of functional equivalence. In previous work, a lowering algorithm for the ATL language was developed and formally verified to generate imperative, C-style loop nests [Liu et al. 2024a, 2022]. We extend this lowering to accommodate the new scatter and coiteration constructs and two reshape operators, shear and unshear. The lowering of coiteration is currently trusted. We use this extended lowering implementation to evaluate our framework by optimizing a set of example programs computing over sparse operands stored in various formats. After compiling the resulting ATL programs to C, we find that ATL-generated sparse kernels achieve comparable performance to the Tensor Algebra Compiler (TACO) framework [Kjolstad et al. 2017]. Currently, our approach supports deriving programs operating over sparse inputs but only produces dense outputs. Supporting writing sparse output structures in a verified, format-parametric way would require extending our format algebra with verified layout-construction operations, which we leave to future work.

Our formal level-format abstraction is similar to that proposed by Chou et al. [2018]. However, their abstraction was designed primarily to guide code generation across different storage formats, whereas ours is designed as a formal algebraic interface for reasoning about compression and decompression schemes. In TACO's design, level formats expose different sets of operations depending on their capabilities, and the presence or absence of particular operations informs the generated code. While effective for code generation, this design does not provide a uniform interface suitable for formal reasoning. In contrast, our abstraction is mechanized in Rocq and presented as a unified interface consisting of functions together with semantic specifications and theorems that each format instance must satisfy. This structure enables mechanized proofs of soundness for compression and decompression routines across format instances.

The implementation and proof are available open-source and as an artifact[Liu 2026].

$$
\begin{array}{rrcl}
\text{Variable} & x & \in & \mathbb{S} \\
\text{Constant} & i & \in & \mathbb{Z} \\
\text{Index} & I & ::= & x \mid i \mid I + I \mid I - I \mid I \times I \mid \lceil I/I \rceil \mid I/I \mid I\%I \\
\text{Predicate} & p & ::= & \text{true} \mid \text{false} \mid I = I \mid I < I \mid I \leq I \mid p \wedge p \\
\text{Expression} & e & ::= & x \mid [\ p\ ] \cdot e \mid \text{let } x := e \text{ in } e \mid \boxplus_{x=I}^{I} e \mid \\
 & & & \sum_{x=I}^{I} e \mid e \oplus e \mid e[I] \mid e * e \mid e/e
\end{array}
$$

Fig. 1. Core ATL constructs

Concatenate	$e \circ e$
Transpose	e^T
Flatten	flatten e
Split	split I e
Right Pad	pad_r I e
Left Pad	pad_l I e
Right Truncate	trunc_r I e
Left Truncate	trunc_l I e

Fig. 2. Reshape operators

2 ATL Scheduling Framework

ATL is a verified tensor-optimization framework embedded in the Rocq proof assistant. It consists of the ATL language, a library of verified scheduling rewrites used for source-to-source program transformations, and a verified lowering algorithm that compiles programs into efficient, imperative C code [Liu et al. 2024a, 2022].

2.1 Core ATL Language

ATL is a pure, functional tensor-programming language. It comprises a set of core constructs for expressing high-level computation for tensor algorithms, as shown in Figure 1. For example, the ATL program below shows a simple matrix-vector multiplication and the corresponding generated C code. It represents the product between the dense matrix m and dense vector v:

$$\boxplus_{i=0}^{N} \left(\sum_{j=0}^{M} m[i;j] * v[j] \right)$$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    out[i] += m[i * M + j] * v[j];
```

The $\boxplus$ construct in this program represents tensor generation ("gen"), which constructs a tensor by describing each element as a function of its indices. This tensor generation produces a one-dimensional tensor of length N. In this case, the body function is a tensor summation, which sums the values of the inner body function evaluated as a function of its iterating index. Therefore, this program produces a vector where each i-th entry is the dot product between the i-th row of m and the vector v. The ATL framework also includes a lowering algorithm that transforms these high-level tensor programs into C code with imperative loop nests that operate on flattened arrays [Liu et al. 2024a]. Shown above is a dense-matrix-vector-multiplication ATL program and its compiled C program. Note that the tensor-generation and tensor-summation constructs produce the analogous structure of nested for-loops in the C code. Lowering establishes the *computation and storage order* of the program. The computation order is determined by the program's loop-nest structure, which establishes the order in which for-loop indices take on values and when they change relative to each other. The relative storage order is established by the index-access expression of the storage statement at the bottom of the loop nest. This determines the destinations into which elements are written into the output buffer as the loop-nest iterates. However, when optimizing such programs we often want to change compute and storage orders to improve locality.

2.2 Reshape Operators

ATL provides a set of *reshape operators*: combinators that modify the relationship between computation and storage order. Each reshape operator shown in Figure 2 corresponds to a common tensor transformation such as transpose, flatten, and split. These operators are defined in

terms of core ATL constructs and can be unfolded into those definitions to execute natively. Their real purpose, however, is to alter how storage indices are computed during lowering. This effectively transforms the mapping between logical computation and physical memory layout. This effect is best understood through examining specific rules of the lowering algorithm, denoted as a function $\mathcal{L}$ whose select production rules are shown to the right. The arguments to this function are e, the ATL program to lower; o, the pre-allocated output buffer name; a, an assignment operator that is either assignment (=) or reduction (+=); and most significantly, a function θ of type $\mathsf{list}(\mathbb{Z} * \mathbb{Z}) \to \mathsf{list}(\mathbb{Z} * \mathbb{Z})$.

$$\begin{aligned}\mathcal{L}\ e\ o\ a\ \theta := \texttt{match}\ e\ \texttt{with}\ \ldots\\ \mid\ R\ e' \Rightarrow \mathcal{L}\ e'\ o\ a\ (\theta \circ \theta_R)\\ \mid\ \texttt{s} \Rightarrow o[\texttt{flatten}\ (\theta\ [\,])]\ \texttt{a s end}\end{aligned}$$

The final argument is the *reindexer* and is used to construct the ultimate storage-assignment index. An index is represented as a list of integer tuples representing the index and size of each dimension. A reindexer is a function from index to index. A reshape operator θ modifies the reindexer before passing it recursively to $\mathcal{L}$ by composing the input reindexer with its own predefined reindexer function θ_R. At the base case, scalar-expression lowering applies θ to the empty index list to produce the final flattened storage index.

Therefore, while core ATL constructs establish the compute order to generate via loop nests in the target C code, reshape operators inform the lowering of how to reorder storage. This feature is crucial for lowering a high-level, functional tensor language into fast and efficient imperative loop nests. Reshape operators enable programmers to describe computation and storage order separately while optimizing a program in ATL, but in a principled, sound way.

2.3 Scheduling Rewrites

Notably, each reshape operator has an adjoint operator such that composing the two yields the identity function. For example, a right-truncation of a right-pad or a transpose of a transpose should yield the original tensor. These are referred to as *reshape-operator adjoint pairs* [Liu et al. 2022] and are formally proven as theorems in the ATL scheduling framework. These adjoint-pair identities are then applied as optimizing rewrites to transform programs, source-to-source.

A user begins with a kernel written in core ATL and uses adjoint-pair theorems to introduce reshape operators. One operator in a pair is "unfolded" (replaced by its definition), yielding a change in computation order. Meanwhile, the reshape operator left in-place will induce the corresponding change in storage order, affecting the way storage indices are generated.

For example, *tiling* is an optimization to improve cache locality and expose opportunities for parallelism. It does so by turning one loop into two nested loops, dividing the iteration space into smaller blocks, or tiles, that can fit in cache. To perform this optimization, we begin by looking at the following adjoint-pair identity theorem. This theorem requires that the tensor that is being tiled is evenly divisible by the tile size and that the tile size is positive. Assuming these conditions are true, it states that split-ting a tensor and then flatten-ing it will produce the same, original tensor.

$$\textsc{FlattenSplitId}\ \frac{|t|\ \%\ k = 0 \qquad 0 < k}{t = \mathsf{flatten}\ \ (\mathsf{split}\ k\ t)}$$

To tile the matrix-vector-multiplication example program we looked at previously, we rewrite it using this theorem with a tile size of $k = 32$ to attain the following ATL program.

$$\mathsf{flatten}\ \left(\mathsf{split}\ 32\ \left(\boxplus_{i=0}^{N} \sum_{j=0}^{M} m[i;j] * v[j]\right)\right)$$

From here, we unfold the split operator to achieve the desired iteration order, which is a nested, tiled loop structure. We can also leverage the ATL scheduling-rewrite framework to simplify the

loop-nest structure with the knowledge that the bounds are divisible by the tiling factor. Hence, we arrive at the tiled matrix-vector multiplication ATL program and the lowered C shown below.

$$\text{flatten} \boxplus_{i_o=0}^{N/32} \boxplus_{i_i=0}^{32} \sum_{j=0}^{M} m[i_o * 32 + i_i; j] * v[j]$$

```
for (int io = 0; io < N / 32; io++)
  for (int ii = 0; ii < 32; ii++)
    for (int j = 0; j < M; j++)
      out[io * 32 + ii] =
        m[(io * 32 + ii) * M + j] * v[j];
```

While the previous tensor generation had an upper bound of N, it has now been split into two nested tensor generations going up to $N/32$ and 32 respectively. However, without the flatten operator, tensor generations alone in this program would produce a two-dimensional matrix rather than the expected vector output.

3 Computing Over Sparse Structures

The ATL language and scheduling framework focuses on computations over dense tensors. However, if some of the operands in a tensor program are highly sparse, meaning that most of their values are zero, even after optimizations like tiling such programs still perform useless work. To avoid unnecessary computation, sparse tensor operands are often stored in a compressed format structure. Ideally, we would like to start with a program defined on dense operands for simplicity and derive an optimized, functionally equivalent kernel for sparse operands.

3.1 A Concrete Sparse Format

While there are numerous formats used across real-world sparse applications, we begin by examining one widespread, common format: compressed sparse row, or CSR. In CSR, each row of a tensor is compressed, and its zero values are removed. In Figure 3, we show an example 4×5 sparse matrix, alongside the construction of its compressed representation in CSR format. Each row of the matrix is compressed using the function compress_row defined below that returns the remaining nonzero values along with their corresponding column coordinates. The overall CSR compression structure is produced by the function dense_to_CSR shown below. The final positional array is the prefix sum of the row-positional arrays' final entries representing the numbers of nonzeros per row. Therefore in the final row-position pointer array, each i-th entry marks the position in the underlying coordinate and data arrays that belong to the i-th row of the dense tensor. The final coordinate array is constructed by concatenating the rows' coordinate arrays. Likewise the final data array contains the concatenated data-value arrays of the rows, storing the tensor's nonzeros in one flat, contiguous buffer.

```
compress_row v = let enum := zip (range 0 |v|) v in
                 let nonzeros := filter (fun (c, x) ⇒ x != 0) enum in
                 ([0; |nonzeros| ],map fst nonzeros, map snd nonzeros)
dense_to_CSR m = let N := |m| in
                 let pcv := map compress_row m in
                 let poss := fold (fun l a ⇒ l ++ [a + last l]) (map fst pcv) [0] in
                 let crds := concat (map (fst . snd) pcv) in
                 let vals := concat (map (snd . snd) pcv) in
                 (N, poss,crds,vals)
```

We draw a distinction between an index representing the *position* of an element, its physical position in the compressed data layout; and one representing the *coordinate* of an element, its logical position in a tensor's dense representation.

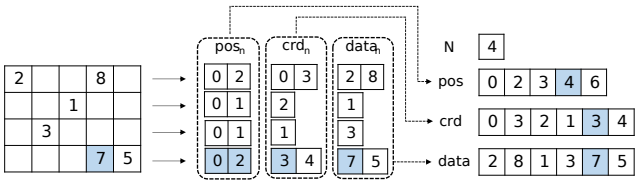


Fig. 3. The derivation of the CSR representation of a matrix. An element of the dense matrix and its corresponding sparse components are highlighted.

3.2 Compression-Decompression as an Adjoint Pair

In many ways, compressed formats can be viewed as a particularly exotic form of storage reordering, and decompression is the natural dual of compression: the optimized pattern for iterating over the compressed data. In this case, a specific compression routine will provide the specification for this storage reordering and how to reinterpret it faithfully, given its metadata, into the original dense tensor representation.

Rather than implementing a new reshape-operator pair for each sparse format and necessitating individual adjoint-theorem proofs, we take a parametric approach. In this work, we propose the definition of a new reshape-operator pair, compress and decompress. We define a general, levelized (per-dimension), modular abstraction for describing sparse formats, similar to the work of Chou et al. [2018]. We use this level-format abstraction to construct multidimensional compression formats compositionally and implement a format-agnostic way to generate code to compute over sparse structures. In this case, compress will be parameterized by a list of level-format instances, F, compositionally describing a multidimensional tensor-compression format.

We not only define a level-format abstraction, but we also formalize it with a set of properties that must be proven for instances of these level formats. These properties ensure soundness of these structures and well-behaved decompression schemes. Therefore, not only are we able to derive computations over sparse structures without case analysis over each potential format, we are also able to prove soundness for this adjoint theorem once and for all in a format-agnostic way.

Let us revisit the matrix-vector-multiplication program in the case where m is a sparse matrix. To derive an optimized sparse program, we begin by rewriting with the compress-decompress adjoint theorem around the tensor operand we want to compute over in a sparse format. Applying this adjoint-theorem rewrite, we arrive at the following program.

$$\boxplus_{i=0}^{N} \sum_{j=0}^{M} (\text{decompress}\ \ (\text{compress}\ F\ m))\ [i;j] * v[j]$$

Note that in general we never intend to call compress at runtime. Instead, we abstract out the call to compress into a binding as follows.

$$\text{let}\ [m_1;\ldots;m_n]\ := \text{compress}\ F\ m\ \text{in}\ \boxplus_{i=0}^{N} \sum_{j=0}^{M} (\text{decompress}\ \ ([m_1;\ldots;m_n]))\ [i;j] * v[j]$$

Rather than evaluating this let-binding as part of our program, we require that these let-bound variables are passed in as function input to the let-binding body, which is the newly derived sparse kernel. In compilation, the kernel will be the body of this let-binding; and the bound metadata index structures are interpreted as free variables taken in as input to the program, similar to the vector argument. Furthermore, when F is instantiated with specific level formats, we know the types of each of these metadata objects. For example, if we instantiate F with the level formats to express CSR, the metadata index structures $m_1;\ldots;m_n$ returned from compress will be the integer and value arrays pos, crd, and data like the ones shown in Figure 3. Hence we can unfold decompress to introduce a new compute order that iterates over the compressed metadata-index structures instead

of the full space of dense coordinates and arrive at the following program.

$$\boxplus_{i=0}^{N} \sum_{p=\text{pos}[i]}^{\text{pos}[i+1]} \text{data}[p] * v[\text{crd}[p]]$$

The outer tensor generation remains the same since each row in CSR is still represented in its compression. However, the bounds of the inner tensor summation are different. Rather than iterating over the full size of the dense dimension, it considers only the nonzero values stored in the compressed format by iterating over the row-position array. As a result, there is an access into the underlying data array by position rather than an access into m by coordinate. To access the vector v, the positional access index p must be converted into the equivalent logical, dense coordinate. This translation is performed by an access into the coordinate-index array stored as part of CSR.

4 Extending ATL

In this section we go over the extensions we made to the ATL framework to support new operators and iteration patterns that will appear in ATL programs after the introduction of sparse formats. Extensions include a set of new core constructs to the ATL language in order to accommodate the operations we will need to express computation over sparse structures and a new pair of reshape operators that extends the set of expressible formats.

4.1 New Core ATL Constructs

We begin by introducing new core ATL language constructs used to express sparse iterations and computation over sparse metadata-index structures. These constructs are instrumental in being able to express decompress in a format-agnostic way, as well as generally enabling us to describe efficient traversal of sparse spaces in ways that the existing ATL constructs could not.

4.1.1 Integer and Boolean Arrays. In the syntax specification, we extend the grammar of index and Boolean expressions to include access into integer arrays as follows:

$$\text{Index Expression} \quad I ::= \ldots \mid x[I] \qquad \text{Predicate} \quad p ::= \ldots \mid x[I]$$

These expressions will be used in programs computing over certain sparse index-metadata structures. Recall from the CSR-formatted matrix-vector-multiplication program that such expressions are needed to capture accesses into the position and coordinate arrays. Similarly, Boolean array access can appear in the predicate expressions of Iverson brackets in programs. The Iverson bracket, inspired by APL, guards an expression by returning its value when its predicate is true, otherwise returning a tensor of zeros [Iverson 1962]. With this extension, we can guard the values of expressions based on separate index-metadata structures.

4.1.2 Scatter. We implement a new scatter construct—an iterating construct similar to tensor generation and tensor summation, denoted by two crossed arrows (⤫). The scatter construct allows us to iterate over a the sparse coordinate space of one tensor while writing and relating it into the dense dimension of the output. The denotational semantics of the scatter construct are shown in Figure 5. The scatter begins with a zero-filled tensor of length N with the proper shape and dimensionality as typed by the expression e, which is a function of i. Then the index i for every value between lo inclusive and hi exclusive will update this tensor at index $f\ i$ with a value $e\ i$. In this sense it is similar to tensor generation, although rather than storing the value $e\ i$ at index i, it adds it at a "scattered" index $f\ i$. The rewrite rule used to introduce scatter into ATL programs, RWScat, is shown in Figure 6.

We extend the ATL lowering algorithm developed by Liu et al. [2024a] with a new production rule for the scatter operator as shown in Figure 4. This rule generates a for-loop with bounds

$$
\begin{aligned}
&\mathcal{L}\ e\ o\ a\ \theta := \texttt{match}\ e\ \texttt{with}\ \ldots \\
&\quad \mid \bigtimes_{i=\mathrm{lo}}^{\mathrm{hi}\le N} \{f\}\ e' \Rightarrow \texttt{for}\ (\texttt{int i = lo; i < hi; i++})\ \{\ \mathcal{L}\ e'\ o\ (+=)\ (\lambda\ \mathrm{idx}.\ \theta\ ((f\ i, N) :: \mathrm{idx}))\ \} \\
&\quad \mid \bigtimes_{p_1=\mathrm{lo}_1<\mathrm{hi}_1,\ldots,p_n=\mathrm{lo}_n<\mathrm{hi}_n}^{\mathrm{arr}_1,\ldots,\mathrm{arr}_n} [\ P\ ] \cdot e' \Rightarrow \texttt{int}\ \mathrm{p}_1 = \mathrm{lo}_1; \ldots\ \texttt{int}\ \mathrm{p}_n = \mathrm{lo}_n; \\
&\qquad\qquad \texttt{while}\ (\mathrm{p}_1 < \mathrm{hi}_1\ \&\&\ \ldots\ \&\&\ \mathrm{p}_n < \mathrm{hi}_n)\ \{ \\
&\qquad\qquad\quad \texttt{int m} = \texttt{min}(\mathrm{arr}_1[\mathrm{p}_1], \ldots, \mathrm{arr}_n[\mathrm{p}_n]); \\
&\qquad\qquad\quad \texttt{if}\ (\mathrm{P})\ \{\ \mathcal{L}\ e'\ o\ (+=)\ \theta\ \} \\
&\qquad\qquad\quad \mathrm{p}_1\ \texttt{+=}\ (\texttt{int})\ (\mathrm{arr}_1[\mathrm{p}_1] == \texttt{m}); \ldots\ \mathrm{p}_n\ \texttt{+=}\ (\texttt{int})\ (\mathrm{arr}_n[\mathrm{p}_n] == \texttt{m});\ \}\ \texttt{end}
\end{aligned}
$$

Fig. 4. Lowering of scatter and coiteration constructs

$$
\left[\!\!\left[\bigtimes_{i=\mathrm{lo}}^{\mathrm{hi}\le N} \{f\}\ e \right]\!\!\right] = \mathrm{fold}\ (\lambda\ l\ i.\ l[f\ i \leftarrow [\![e\ i]\!] + l[f\ i]])\ [\mathrm{lo}, \mathrm{lo}+1, \ldots, \mathrm{hi}-1]\ (\mathrm{repeat}\ 0\ N)
$$

$$
\begin{aligned}
\left[\!\!\left[\bigtimes_{p_1=\mathrm{lo}_1<\mathrm{hi}_1,\ldots,p_n=\mathrm{lo}_n<\mathrm{hi}_n}^{\mathrm{arr}_1,\ldots,\mathrm{arr}_n} [\ P\ ] \cdot e \right]\!\!\right] = {} & \mathrm{fold}\ (\lambda\ (r, \mathrm{ps})\ \_\ .\ (\mathrm{incr}\ \mathrm{ps}\ [\mathrm{arr}_1; \ldots; \mathrm{arr}_n], \\
& r \oplus [\ \mathrm{fold}\ (\&)\ ([\mathrm{ps}[0] < \mathrm{hi}_1; \ldots; \mathrm{ps}[n-1] < \mathrm{hi}_n])\ (P\ \mathrm{ps})\ ] \cdot [\![e\ \mathrm{ps}]\!])) \\
& (\mathrm{repeat}\ 0\ (\Sigma_{i=1}^{n} \mathrm{hi}_i - \mathrm{lo}_i))\ (0, [\mathrm{lo}_1; \ldots; \mathrm{lo}_n])
\end{aligned}
$$

Fig. 5. Scatter and coiteration denotational semantics

$$
\textsc{RWCoit}\ \frac{\mathrm{ordered}_c^n\ c_1 \qquad \mathrm{unique}_c^n\ c_1 \qquad \mathrm{ordered}_k^m\ c_2 \qquad \mathrm{unique}_k^m\ c_2}{\sum_{i=0}^{n} \sum_{j=k}^{m} [\ c_1[i] = c_2[j]\ ] \cdot d_1[i] * d_2[j] = \bigtimes_{i=0<n, j=k<m}^{c_1, c_2} [\ c_1[i] = c_2[j]\ ] \cdot d_1[i] * d_2[j]}
$$

$$
\textsc{RWScat}\ \frac{\forall i.\ 0 \le i < m \to 0 \le \mathrm{rdx}\ i < n}{\boxplus_{i=0}^{n} \sum_{j=0}^{m} [\ \mathrm{rdx}\ j = i\ ] \cdot e\ j = \bigtimes_{p=0}^{m \le n} \{\mathrm{rdx}\ i\}\ e\ p}
$$

Fig. 6. Rewrite rules for introducing scatter and coiteration

matching the iterating bounds of lo and hi. The body of this for-loop is the lowering of the body of the scatter, but it builds upon the index by adding a tuple of the scattered version of its iterating index $(f\ i)$ and the full size of the scattered dimension N.

4.1.3 Coiteration. The coiteration operator denoted by $\bowtie$ enables joint iteration over multiple integer arrays in ATL, enabling efficient traversal of multiple integer arrays and accumulation of results based on values at corresponding index positions that satisfy a given predicate. The efficiency of coiteration comes from the knowledge that the integer arrays are *locally ordered*. Instead of scanning each array to find matching and predicate-satisfying entries during each iteration, the construct performs an n-finger merge [Kjolstad et al. 2017]. As a result, all array indices are advanced in a coordinated manner so that each element is visited at most once.

The denotational semantics for coiteration is shown in Figure 5. The construct takes as input n integer arrays $\mathrm{arr}_1, \ldots, \mathrm{arr}_n$, along with n lower and upper bounds, specifying the positional index ranges for each array. It also takes a body e and predicate P, both functions over the array positions. At each step of the iteration, if the predicate P holds for the current positions and all positions are within bounds, the application of e is accumulated. Then, among the current positions, those pointing to the smallest array value are incremented using the incr function (referenced in Figure 5). Because at least one position is advanced in each iteration, the number of evaluations of the body expression is bounded by the combined positional spans of the input arrays, or $(\Sigma_{i=1}^{n} \mathrm{hi}_i - \mathrm{lo}_i)$. A common rewrite rule RWCoit used to introduce coiteration into ATL programs is shown in Figure 6. Note that it requires that the input integer arrays being coiterated must contain elements that are ordered and unique within the coiterating bounds.

The corresponding lowering rule shown in Figure 4 produces a while loop. Each integer array is associated with a pointer initialized to its corresponding lower bound outside this loop, which continues as long as every pointer remains below its corresponding upper bound. Inside the loop, the current minimum value among the pointed-to elements of the arrays is determined. The construct then checks whether the current positions satisfy the predicate. If so, the body expression is evaluated, and its result is accumulated. Finally, each pointer that currently points to the minimum value is incremented before the next iteration begins.

4.2 New Reshape Operators

We introduce a new pair of reshape operators, shear and unshear, to expand the set of multidimensional sparse formats expressible in the ATL framework. Later, we will show how these operators are used to derive computations involving a sparse tensor format known as DIA (Section 8).

The shear operator constructs a new tensor where each row stores a diagonal of the original tensor. Since many diagonals of a tensor are not the same length, the shear operator pads these diagonals to the length of the longest diagonal (the main diagonal of the tensor) with zero values. unshear reconstructs a tensor with n rows and m columns by storing an input tensor's rows as diagonals, truncating the lengths of these rows as necessary to produce a $n \times m$-size output. The definitions for these operators are shown below.

$$\begin{aligned}\text{shear } t := \quad & \text{let } D := \min |t|\ |t[0]| \text{ in } \boxplus_{d=0}^{|t|+|t[0]|-1} \boxplus_{r=0}^{D} \\ & [\ 0 \le \max 0\ (n - D - (\max 0\ (d - D + 1))) + r - n + 1 + d < |t[0]|\ ]\cdot \\ & t[\max 0\ (n - D - (\max 0\ (d - (D-1)))) + r; \\ & \quad \max 0\ (n - D - (\max 0\ (d - D + 1))) + r - n + 1 + d]\end{aligned}$$

$$\text{unshear } n\ m\ t := \boxplus_{i=0}^{n} \boxplus_{j=0}^{m} t[j - i + n - 1; i - \max 0\ (n - (\min n\ m) - (\max 0\ (j - i + n - 1 - (\min n\ m - 1))))]$$

We prove the following adjoint identity theorem for these reshape operators.

$$\textsc{UnshearShearId}\ \frac{}{t = \text{unshear } |t|\ |t[0]|\ (\text{shear } t)}$$

We extend the ATL lowering algorithm to accommodate shear and unshear by following the reshape-operator lowering form shown earlier with the following corresponding reindexers.

```
θ_shear idx := match idx with
        | (v, d)::(v', d')::idx' ⇒
                (v' − v + d − 1, d + d' − 1)::
                (v − (max 0 ((d − min d d') − (max 0 (v' − v + d − min d d')))), min d d')::idx'
        | _ ⇒ idx end
θ_unshear N M idx := match idx with
        | (d, n)::(r, m)::idx' ⇒
                (max 0 (N − (min N M) − (max 0 (d − min N M + 1))) + r, N)::
                (max 0 (N − (min N M) − (max 0 (d − min N M + 1))) + r − N + 1 + d, M)::idx'
        | _ ⇒ idx end
```

5 Level-Format Abstraction

The core of our formalization of sparse formats is the level-based abstraction. The abstraction encapsulates how to apply an encoding routine to one dimension of a tensor and produce a corresponding metadata-index structure for an abstract level format F. It also includes functions detailing how to access and interpret the underlying data within a format's layout. Our level-format abstraction includes not just the level-format functions but also properties of their semantic specifications. These ensure that the abstraction functions encode a compression format that can

$$\text{encode}_F : \text{list } X \to M_F * \text{list } X$$
$$\text{append}_F : M_F \to M_F \to M_F$$
$$\phi_F : M_F$$
$$\text{well\_formed}_F : M_F \to \text{Prop}$$
$$\text{crd\_pos}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_c \to \text{option } \mathbb{Z}_p$$
$$\text{pos\_crd}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_p \to \text{option } \mathbb{Z}_c$$
$$\text{pos\_pos}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_p \to \text{option } \mathbb{Z}_p$$
$$\text{pos\_bounds}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_p * \mathbb{Z}_p$$
$$\text{valid}_F : M_F \to \mathbb{Z}_p \to \text{bool}$$

(a) Format-abstraction definitions

$$\textsc{MonoidStruct}\ \frac{}{(\text{well\_formed}_F\ M_F, \phi_F, \text{append}_F)\ \text{forms a monoid}}$$

$$\text{Let } (m_F, t') := \text{encode}_F\ t \text{ in}$$

$$\textsc{EncodeWellFormed}\ \frac{}{\text{well\_formed}_F\ m_F}$$

$$\textsc{CrdPosGet}\ \frac{\text{crd\_pos}_F\ m_F\ 0\ c = p}{t[c] = t'[p]}$$

$$\textsc{CrdPosNoneGet}\ \frac{\text{crd\_pos}_F\ m_F\ 0\ c = \text{None}}{t[c] = 0}$$

$$\textsc{InvalidPosGet}\ \frac{\neg\ \text{valid}_F\ m_F\ 0\ p}{t'[p] = 0}$$

(b) Properties of encoding M_F

$$\textsc{PosCrdInj}\ \frac{\text{valid}_F\ m_F\ p'\ p_1 \qquad \text{valid}_F\ m_F\ p'\ p_2 \qquad \text{pos\_crd}_F\ m_F\ p'\ p_1 = \text{pos\_crd}_F\ m_F\ p'\ p_2}{p_1 = p_2}$$

$$\textsc{CrdPosInv}\ \frac{\text{crd\_pos}_F\ m_F\ p'\ c = p \qquad \text{valid}_F\ m_F\ p'\ p}{\text{pos\_crd}_F\ m_F\ p'\ p = c} \qquad \textsc{PosCrdInv}\ \frac{\text{pos\_crd}_F\ m_F\ p'\ p = c \qquad \text{valid}_F\ m_F\ p'\ p}{\text{crd\_pos}_F\ m_F\ p'\ c = p}$$

$$\textsc{CrdPosNone}\ \frac{\text{crd\_pos}_F\ m_F\ p'\ c = \text{None} \qquad i \in \text{pos\_bounds}_F\ m_F\ p' \qquad 0 \le c < \dim m_F}{\text{pos\_crd}_F\ m_F\ p'\ i \neq\ c}$$

(c) Properties of position-coordinate translation functions

$$\textsc{CrdBound}\ \frac{\text{valid}_F\ m_F\ p'\ p \qquad \text{pos\_crd}_F\ m_F\ p'\ p = c}{0 \le c < \dim m_F} \qquad \textsc{PosBound}\ \frac{\text{crd\_pos}_F\ m_F\ p'\ c = p}{p \in (\text{pos\_bounds}_F\ m_F\ p')}$$

(d) Properties of coordinate and position bounds

$$\begin{array}{lll} \text{Let } \phi s & := & \text{map } (\text{fst} \circ \text{encode}_F)\ t \text{ in} \\ \text{Let } \delta & := & \text{map } (\text{snd} \circ \text{encode}_F)\ t \text{ in} \\ \text{Let } \Phi & := & \text{fold append}_F\ \phi s\ \phi_F \text{ in} \end{array}$$

$$\textsc{CrdPosApp}\ \frac{}{\text{crd\_pos}_F\ \Phi\ p\ c = \text{crd\_pos}_F\ (\phi s[p])\ 0\ c}$$

$$\textsc{BoundsGet}\ \frac{}{\begin{array}{l}\text{snd } (\text{pos\_bounds } (\text{encode}_F\ t[p]))\ 0 - \text{fst } (\text{pos\_bounds } (\text{encode}_F\ t[p]))\ 0 = \\ \text{snd } (\text{pos\_bounds}_F\ \Phi\ p) - \text{fst } (\text{pos\_bounds}_F\ \Phi\ p)\end{array}}$$

$$\textsc{ValidGet}\ \frac{}{\text{valid}_F\ \Phi\ (\text{pos\_pos}_F\ \Phi\ p'\ p) = \text{valid}_F\ (\text{fst } (\text{encode}_F\ t[p']))\ 0\ p}$$

$$\textsc{AppGet}\ \frac{}{(\text{concat } \delta)[\text{pos\_pos}_F\ \Phi\ c\ (\text{crd\_pos}_F\ \Phi\ c\ p)] = t[c; p]}$$

(e) Properties of append_F

Fig. 7. Level-format abstraction for a format F. Here X represents the tensor element type.

be reinterpreted into the original dense tensor. For each level-format instance, these functions are implemented and their properties proven. The algebraic signature for this abstraction presented in Figure 7 is realized in the Rocq proof assistant.

5.1 Level-Encoding Functions and Index Structures

As part of the abstraction, a level format F must provide certain definitions that define the format encoding. These definitions are listed in Figure 7a.

They include the level-encoding function encode_F that takes a tensor of rank at least 1 and returns a tuple of the remaining elements after compression and a corresponding metadata-index structure of type M_F. This type M_F is also specified per level format.

The abstraction also requires an operation append_F, which combines two metadata index objects into one. This function is folded over the metadata-index structures produced when compression is mapped over elements in one dimension. The resulting metadata-index structure is then used to describe how to access and decode the dimension as a whole.

The abstraction also includes a specific metadata-index structure ϕ_F, which is meant roughly to represent an "empty" index structure.

The final definition regarding the encoding-metadata-index structures that must be provided is a well-formedness predicate defined on elements of type M_F.

A format must also provide functions to access a metadata-index structure and meaningfully iterate over it. The indices used for access and iteration are integers, though they could represent (logical) coordinates or (physical) positions. To make this difference apparent in the type signatures, we will indicate the type of a positional index as $\mathbb{Z}_p$ and a coordinate as $\mathbb{Z}_c$.

The format function crd_pos is a partial function that converts coordinates to positions. In addition to the integer coordinate, this function takes in a metadata-index structure and a position argument. This position represents an offset in the metadata structure in case the dimension being decoded is an inner dimension (recall that the metadata structure may be a combination structure of several encoded elements within this dimension). We will call this kind of positional argument the parent position. Similarly, a partial function pos_crd must be provided to be able to convert from a position to its corresponding coordinate.

The pos_pos function takes a positional index in one dimension of a metadata-index structure and returns the positional index in the next dimension. This transition corresponds to the start of the structure describing the elements of the tensor element, representing an access from a higher dimension into a lower one but through sparse positions rather than coordinates.

Relatedly, the function pos_bounds must return the span of positions corresponding to a tensor element in the underlying dimension, as a function of a position and a metadata-index structure. The span of these positions is returned as a tuple of integers denoting the lower and upper bound of the relevant positions.

Finally, depending on the level format, not every position within the positional bounds returned by pos_bounds_F necessarily corresponds to a nonzero tensor element. Therefore, a level format also provides a valid function that takes a metadata-index structure and position and returns whether or not that position is valid, or actually contains an element to be decoded further.

5.2 Level-Encoding and Index-Structure Properties

Given the abstraction index-structure and function definitions, the level-format abstraction also requires these definitions to uphold the properties stated in Figure 7b.

The first property MonoidStruct requires that the set of well-formed elements of M_F must be closed under append_F, an associative operation over well-formed elements of M_F with ϕ_F as an identity element. In other words, well-formed M_F must form an algebraic monoid under append_F.

The rest of this set of properties listed in Figure 7b establishes how we expect the access and iteration functions to interact with a metadata-index structure produced by encode_F.

First, EncodeWellFormed states that the encoding procedure must produce metadata index structures that are well-formed under the definition of this abstraction. Next, the property CrdPosGet states if a coordinate c is encoded to have a position p by m_F, then accessing into the original tensor t at index c would be equivalent to accessing into the compressed data output t' at index p. Similarly, CrdPosNoneGet states if coordinate c does not have a position represented in m_F, then accessing into the dense tensor t at index c must be a tensor of zeros. In other words, the element in t at index c was compressed out. Likewise, InvalidPosGet states if p is not a valid position within m_F, then accessing into the compressed data t' at index p will return a tensor of zeros.

5.3 Properties of Coordinate-Position Translation

The coordinate-to-position and position-to-coordinate translation functions of a level format must also fulfill key properties listed in Figure 7c.

The first property CrdPosInv states that if a coordinate c is translated to a position p, and p is a valid position within a metadata-index structure m_F, then converting p back to a coordinate through m_F should yield c. Likewise, PosCrdInv states that if we begin with a valid position p, and converting p to a coordinate yields c, then the position translation of c should return p. Additionally, PosCrdInj also enforces injectivity on these translation functions. If there are two valid positions p_1 and p_2 under an index structure m_F, and they translate to the same coordinate, then p_1 must equal p_2. Finally, CrdPosNone states if a coordinate c does not translate to any position in m_F, then no position p between the positional bounds established by m_F translates to coordinate c.

5.4 Properties of Coordinate and Position Bounds

The properties in Figure 7d constrain the range of values that the indices translated through the metadata-index structure can take on. These bounds ensure that accessing the dense and sparse structures through these translation functions will always be in-bounds. These properties pertain to any well-formed metadata-index structure m_F and also any parent position p', as this index structure may result from the appending of several index structures.

The first property CrdBound states that if p is a valid position and translates to a coordinate c through some metadata structure m_F, then c must be a nonnegative integer less than the size of the tensor dimension encoded by m_F. The next property PosBound enforces a similar bound on positions that result from translation. If a coordinate c translates to a position p, then p must be between the positional bounds established by pos_bounds for this index structure.

5.5 Properties of append_F

The set of properties in Figure 7e pertain to the append function that combines metadata-index structures for a given format. In these properties we consider metadata-index structure Φ that is the aggregate structure resulting from the fold of append_F across a list of metadata-index structures, ϕs. Specifically, ϕs is obtained from mapping encode_F over the elements of a tensor t. Note the implication that t is at least two-dimensional. This aggregated metadata-index structure, Φ, represents every element in each "row" of t. These properties defining append_F allow us to decompose a function call at some position p indexing through Φ into a function call to the p-th metadata-index structure in ϕs instead. In essence, we may "access" within a sparse tensor dimension and focus in on subdimensions.

The first property CrdPosApp states that a coordinate c translated through Φ with a parent position of p is equivalent to the position translation of coordinate c from the p-th index structure of ϕs with a new parent-position offset of 0.

Next, BoundsGet states that the positional range encoded by Φ at a parent position p must be the same as the positional range obtained by directly encoding the p-th element of tensor t.

Similarly, ValidGet states if a position p is valid within Φ with a parent position p', then position p must be valid in the index structure produced by directly encoding the element in t.

Finally, AppGet states that append_F maintains the positional properties that align with the underlying encoded data described by the metadata structures. The access into a concatenated collection of sparse data is analogous to an access into a vector produced from flattening a matrix. In other words, accessing into this concatenated list at a position offset from some parent position c through Φ and position p corresponding to a position within an element of that dimension is equivalent to the two-dimensional access of c and then p into the original dense tensor t.

6 Level-Format Instances

In this work, we provide proofs and implementations for a small set of formats as instances of the level-format abstraction. We chose these formats to demonstrate expressivity and compositionality.

6.1 Scalar

The scalar format, denoted as s when passed as an argument to compress, is a trivial format, only applicable as a degenerate compression format onto scalars, or zero-rank tensors. The following shows the scalar instance definition of the level-format abstraction.

$$\mathsf{M_s} := \mathsf{list}\ \mathbb{R} \quad \mathsf{encode_s}\, r := ([r], r) \quad \mathsf{append_s}\ m_1\ m_2 := m_1 \mathbin{+\!\!+} m_2$$

The scalar format must be the final format in the format-list argument to compress as it represents the data payload beneath all the metadata-index structures produced from composing compression. Therefore, the scalar format type is a list of scalar values that represents the data. The encode function for the scalar format is implemented to encapsulate the scalar value as a singleton list. We expect the degenerate case of compressing a zero-rank tensor using the scalar format to result simply in an array containing the scalar value. Finally, the append function is defined as concatenation.

6.2 Dense

The dense format is denoted as d when passed as an argument to compress. This format represents a dimension that will remain uncompressed and retain all of its original elements. The definitions of this instance are shown below.

$$\mathsf{M_d} := \mathbb{Z} \quad \mathsf{encode_d}\ t := (|t|, t) \quad \mathsf{append_d}\ m_1\ m_2 := m_1$$

The only information in the metadata-index structure that needs to be carried by the dense format is the original dimension size. Therefore a metadata index value is an integer. Additionally, since the dense format retains all tensor elements, the data returned by encode will just be the original tensor. The index data returned is the dimension size or simply the length of the input tensor. Therefore, the dense format's encode just returns the input tensor and its size. Given two dimension sizes as metadata structures, the append function will return the first one. This convention is reasonable: since we expect only to be appending together metadata-index structures of tensor elements of the same dimensionality within one consistent tensor, we can assume that the dimension sizes stored in these index structures will match.

6.3 Bitmask

The bitmask format will be denoted as b. This format, like the dense format, retains all elements of a tensor. However, the bitmask format additionally stores a bitvector that encodes whether the tensor element at each corresponding position is a nonzero.

$$\mathsf{M_b} := \mathsf{list}\ \mathbb{B} * \mathbb{Z} \quad \mathsf{encode_s}\, t := (\mathsf{map}\ (\neq\ 0)\ t, |t|, t) \quad \mathsf{append_s}\ m_1\ m_2 := (\mathsf{fst}\ m_1 \mathbin{+\!\!+} \mathsf{fst}\ m_2, \mathsf{snd}\ m_1)$$

For this level format, the metadata-index structure type will store this bitvector as a list of Booleans and the original dimension size as an integer. The bitmask format's encode function returns the input tensor with no values removed along with its bitvector. To construct this bitvector, we map over the input tensor an indicator returning a Boolean denoting whether that element is nonzero. The append function for combining bitvectors will concatenate them. Additionally, like the dense level format, combining metadata-index structures will only be well-defined for tensors of the same shape, so we expect the input index structures to agree on dimension size.

6.4 Hash

The hash format compresses one dimension of a tensor and stores a hash array of the coordinate-index entries of the nonzero elements. This hash array has a fixed width n and must be able to

accommodate all remaining tensor elements in order to be sound. The hash format is parametrized by this width n as well as a hash function f that maps integers to integers. The parameterized hash format can be denoted as h n f and defined as follows. The metadata-index structure for the hash level format must contain the hash integer array, the hash function, the width of the hash arrays, and the original size of the dimension.

$$\mathsf{M_h} := \mathsf{list}\ \mathbb{Z} * \mathbb{Z} * \mathbb{N} * (\mathbb{Z} \to \mathbb{Z}) \quad \mathsf{encode_h} := (\mathsf{hash\_level}\ t, \mathsf{filter}\ t) \quad \mathsf{append_h}\ m_1\ m_2 := m_1 \mathbin{+\!\!+} m_2$$

The following function is a helper used to implement the construction of the hash array, by taking a coordinate c and inserting that coordinate and a corresponding tensor element at either its hashed position or searching for the closest available position without an existing entry.

```
insert n f c x h := match n with
  | 0 ⇒ h (* no room in hashmap *)
  | S n' ⇒ if (fst (h[( f c + |h| - n) % |h| ]) == -1)
    then h[( f c + |h| - n) % |h| ← (c,x)] else insert n' f c x h end
```

To begin, the hash array is entirely empty with index entries of -1 mapped to zero-valued tensors. This definition is provided for the empty metadata-index structure, ϕ_h, for the hash format as well.

$$\phi_{\mathsf{h}} := (\mathsf{repeat}\ (-1, 0)\ n, N, n, f)$$

The encoding function starts with an empty hash array. Then folding over a given tensor, it calls insert to insert each nonzero element along with its dense coordinate into the hash array. We define this function hash_level as follows.

```
hash_level t := snd (fold (fun (c, a) x ⇒
                  (c+1, if x == 0 then a else insert |a| f c x a)) t (0, (repeat (-1, 0) n))
```

Similarly to previous formats, the append function is well-formed only if its arguments are of the same format. For the hash format, this means that the hash widths as well as the hash functions are the same. Therefore, we concatenate together the two arrays and adopt the same hash function. Note that we need not construct a new hash function for this larger map, because all the level-abstraction access functions take a parent-position argument. Thus all hashed indices will translate properly to the offset within the proper, original bucket.

6.5 Compressed

The compressed format removes zero elements in one dimension of a tensor and stores two integer arrays as its metadata-index structure. This level-format instance is defined as follows.

$$\begin{aligned}
&\mathsf{M_c} := \mathsf{list}\ \mathbb{Z} * \mathsf{list}\ (\mathsf{list}\ \mathbb{Z}) * \mathbb{Z} * (\mathbb{Z} \to \mathsf{list}\ \mathbb{Z}) \\
&\mathsf{encode_c}\ t := ([0; |\mathsf{filter}\ t|], \mathsf{comp}\ f\ t, |t|, f, \mathsf{filter}\ t) \\
&\mathsf{append_c}\ m_1\ m_2 := \mathsf{app}\ (\mathsf{fst}\ m_1)\ (\mathsf{fst}\ m_2)\ (\mathsf{snd}\ m_1)\ (\mathsf{snd}\ m_2)
\end{aligned}$$

The first integer array has entries marking the position where each corresponding segment in the underlying data array begins. This array we call the position array, as it stores starting positions for these segments. So the initial call to $\mathsf{encode_c}$ will always return the position array $[0; n]$, where n is the number of tensor elements remaining.

The second integer array stores the coordinates corresponding to the tensor elements remaining after compression. In this work we implement an augmented compressed format. Rather than storing the coordinates directly, we parameterize the format with an injective function f of type $\mathbb{Z} \to \mathsf{list}\ \mathbb{Z}$. For each nonzero tensor element, we apply this function to convert its coordinate into a new representation and store these in an array in the metadata-index structure.

Overall the compressed format metadata-index structure will store the position-index array, the converted-coordinate array, the coordinate-conversion function, and the dimension size. Technically, this format also requires a proof term showing that the coordinate-conversion function is injective

over the domain established by the dimension size. However, this proof term can be automated, and we omit it from the type definition for simplicity's sake.

To define the encode function for this format, we define the following helper function to construct the proper coordinate structures for the compressed format. This function returns a list of the converted coordinates of the elements that remain after compression.

```
comp f t i := map (fun x ⇒ f (snd x)) (filter (fun x ⇒ fst x != 0) (zip t (range 0 | t|) ))
```

As mentioned before, the append function of the level-format abstraction is only meant to be applied between metadata-index structures of the same level format. Therefore for the compressed format we expect the inputs to have the same coordinate-conversion function and dimension size. To preserve the coordinate-index information, we can concatenate together converted coordinate arrays. To construct the aggregated position array, the latter metadata-index structure being appended must be incremented by the number of positions encoded in the first metadata-index structure to reflect the coordinate arrays being concatenated. This number of positions is the length of the metadata structure's converted coordinate array, or the *last* element of its position array. Therefore we can implement the append$_c$ level-format function as follows.

```
app p1 p2 c1 c2 := (p1 ++ (map (fun x ⇒ x + last p1) (remove_last p2)) ,  c1++c2)
```

7 Multidimensional Compression and Decompression

Using the level-format-abstraction interface functions, we can describe multidimensional compression and decompression schemes in a format-agnostic way. Since each level format denotes an encoding scheme on a given dimension of a tensor, we can define compression and decompression recursively over a tensor's dimensions. Compression takes in a rank-n tensor and a list of n level formats, returning a stack of n metadata-index structures, while decompression reverses that flow.

7.1 Multidimensional Compression

The function compress is parameterized by a list of level formats that have been implemented as instances of the formal abstract interface defined previously and applied on a tensor t. The function implementation is shown below.

```
compress F t := match F with
              | [f1;...; fn ] ⇒ let (m1,t') := encode_f1 t in
                                let metas' :=  map (compress [f2;...; fn]) t' in
                                let metas := fold append_list metas' [φf2;...;φfn] in (m1::metas)
              | [s] ⇒ [ t ] end
```

The length of this format list must be the same as the rank of tensor t, because each format applies to one dimension of the tensor, corresponding to its order in the list. For this format list to be well-defined, we also expect its last level format to be the scalar format, or s, with t being a scalar value, reflected in the base case as shown above. Moreover, this function will return a list of metadata-index structures of the same length as the format list.

The top-level dimension will be compressed according to the encoding routine specified by the first level format in the format list. Then each surviving element in the resulting compressed data will be compressed recursively with the remaining formats. Subsequently, each recursively compressed element returns its own stack of metadata objects. Each of these stacks is elementwise appended to construct one final stack of metadata-index structures, the job of the append_list function. Instead of taking two metadata-index structures and returning a new metadata-index structure, it takes two *lists* of metadata-index structures and returns a list. It does so by calling the

appropriate append_F elementwise between the two lists. Note that this function only makes sense if the ordering of formats of metadata-index structures matches between its two list arguments.

By designing the level-format abstraction to include the compress and append functions without interdimensional dependencies, we can build a format-agnostic, compositional way to define compression over a tensor. More notably, we can build and customize multidimensional tensor-compression formats by composing different level formats without having to build by-hand the reasoning needed to interact with and interpret the compression structures between each dimension.

7.2 Multidimensional Decompression

We are also able to use the level-format abstraction to write a format-agnostic definition of multidimensional decompression. Recall that the output of the multidimensional compress function is a list of metadata-index structures of the same length as the rank of the multidimensional tensor it encodes. Likewise, the multidimensional decompress function will take a list of metadata-index structures to decompress and return a tensor of corresponding dimensionality. It will also take in a parent positional index to keep track of its position within the metadata-index structure of the dimension being decompressed. It is defined as follows.

```
decompress M p' := match M with
          | [m₁;...;mₙ] ⇒ let (lo,hi) := pos_bounds m₁ in
                          ⨉_{p=lo}^{hi≤dim m₁} {pos_crd m₁ p' p} [ valid m ] · decompress [m₂;...;mₙ] (pos_pos m₁ p' p)
          | [ r ] ⇒ r[p'] end
```

Similar to the definition of multidimensional compression, the list of metadata-index structures is only well-formed if the last object is the scalar format's index-structure type. In other words, the final object m_n for a rank-n tensor passed to decompress must be a list of scalar values. When this payload array of values is reached in the base case of multidimensional decompression, we simply perform the access of the positional argument that has been translated throughout each dimensional call to decompress.

7.3 Soundness of Compression and Decompression

Using these two function definitions, we are able to define the top-level soundness theorem on compression and decompression. This theorem states that decompressing the metadata-index structures obtained from composing any properly formed list of level formats to compress a tensor will return the original tensor.

$$\textsc{DecompressCompress}\ \frac{\mathsf{formats\_have\_space}\ F \qquad \mathsf{proper\_formats}\ F\ t}{t = \mathsf{decompress}\ (\mathsf{compress}\ \mathsf{F}\ t)\ 0}$$

The property proper_formats states the length of F matches the dimensionality of the tensor t and that its terminal format is scalar. The formats_have_space property states that any fixed-size level formats (such as hash) in F have enough space for the nonzero entries in that dimension of t.

Rewriting using this adjoint-pair theorem, we can turn a standard ATL program computing over dense tensors into one computing over the sparse metadata-index structures. First, the call to compress can be simplified into the list of metadata-index structures. Just like how in the dense ATL case the programs are parameterized over the input tensors and take them in as arguments, the new sparse program will accept the metadata-index structures in this list as arguments. Then, decompress can be unfolded. We are able to unfold this implementation of decompress into a program purely using core ATL language constructs, including the scattering construct we introduced. In doing so, we introduce a loop to iterate over the sparse positions of the compressed tensor rather than the dense coordinates. Finally, we have arrived at an ATL program computing and iterating

Format	compress Call
CSF/DCSR	compress [c; ...; c; s] t
CSR	compress [d; c; s] t
HASH	compress [h; s] t
Bitmask	compress [b; s] t
CSC	compress [d; c; s] (t^T)
BCSR	compress [d; c; d; d; s] $\left(\boxplus_{i=0}^{k_1} \boxplus_{j=0}^{k_2} ((\text{split } k_1\ ((\text{split } k_2\ t)[j])^T)[i])^T\right)$
CSB	compress [d; d; c; c; s] $\left(\boxplus_{i=0}^{k_1} \boxplus_{j=0}^{k_2} ((\text{split } k_1\ ((\text{split } k_2\ t)[j])^T)[i])^T\right)$
DIA	compress [c; d; s] (shear t)
COO	compress [c; s] (flatten t)
mode-generic	compress [c; d; s] $\left(\text{flatten } \left(\boxplus_{i=0}^{n} t[i]^T\right)\right)$

Fig. 8. Compositional level-format derivation of common multidimensional formats

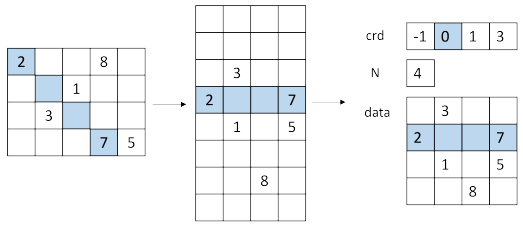
over the compressed sparse structures of some format that is formally proven to be semantically equivalent to the original ATL program computing over dense operands. Moreover, this program can now be further optimized using the existing ATL scheduling-rewrite framework.

8 Integrating Reshape Operators

This set of level-format instances we have provided in this framework is powerful and composes to create numerous multidimensional sparse formats. However, there are some common, idiomatic formats that cannot be expressed through their composition alone. For example, DIA format [Yuan et al. 2010] is a specialized sparse format in which a matrix is stored by its diagonals with empty diagonals being compressed out. No combination of the level-format instances implemented above can produce this layout. In fact, no combination of level-format instances that abide by the algebraic dimensional-isolation property enforced by levelization can produce this layout. The levelization inherently enforces independence between dimensions of a tensor so that information about an element's coordinate or position in one level does not affect its position in another. This separation of concerns ensures modularity and compositional reasoning but limits expressiveness. Indeed, the TACO formats system [Chou et al. 2018] has to introduce level-format instances including Offset, Range, Singleton and a system of format decorators that complicate the format system and weakens levelization in these cases.

Rather than implementing more level formats or breaking the levelization enforced by the current abstraction, we integrate ATL's reshape operators into the sparse framework to achieve the same expressive power through verified rewrites. This combination allows us to represent a broader family of sparse formats, including DIA, without breaking the levelization and modularity properties of the abstraction. Specifically, the addition of shear and unshear reshape operators enables expressing DIA format naturally within ATL.

We conceptualize the conversion of a dense matrix into DIA format via the process shown on the right. We first shear the tensor so that its diagonals become rows, then compress out empty rows while storing the indices so that the main diagonal corresponding to index 0. Each row (formerly a diagonal) remains dense.



To achieve this result formally, we apply the
reshape-adjoint theorem for shear and unshear followed by the decompress–compress adjoint theorem via the series of rewrites shown below. The level formats we pass to compress will be compressed, dense, and then scalar.

$$t \longrightarrow \text{unshear}\ \ (\text{shear}\ t) \longrightarrow \text{unshear}\ \ (\text{decompress}\ \ (\text{compress}\ [c; d; s]\ \ (\text{shear}\ t))\ \ 0)$$

This reshape rearrangement in combination with the level formats passed to the compress function will produce the sparse structures of t stored in DIA format. The structures can be bound in the program like in the unreshaped example cases, and likewise, decompress can be unfolded and rescheduled as a normal ATL program. By combining a minimal set of proven level-format instances with the reshape operators of the ATL framework, we can express numerous multidimensional sparse formats. We derive some of the most common formats including CSF (compressed sparse fiber) or DCSR (doubly compressed sparse row), CSR (compressed sparse row), HASH, bitmask, CSC (compressed sparse column), BCSR (block compressed sparse row), CSB (compressed sparse block), DIA, COO (coordinate list), and mode-generic. These formats and the corresponding calls to compress used to express them are shown in Figure 8.

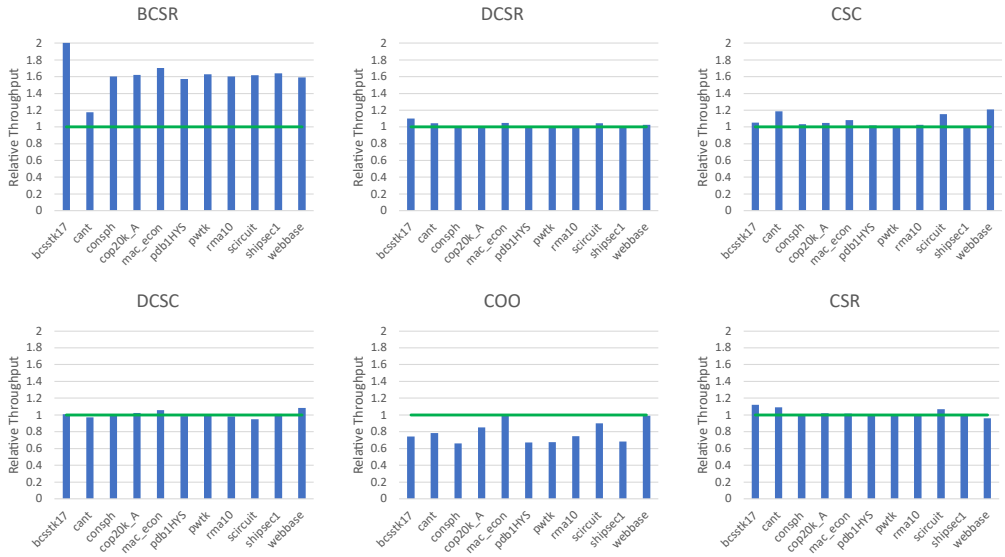
9 Evaluation

We evaluate the ATL-derived sparse kernels against the state-of-the-art sparse tensor compiler, TACO [Kjolstad et al. 2017]. We show that ATL's sparse extension can express a wide range of sparse formats and generate kernels with efficient sparse iteration strategies, achieving desirable performance. Our goal is to obtain performance and format-expressivity comparable to TACO to show we have developed a verified, extensible algebra capable of producing similar code. Our experiments target sparse matrix-vector multiplication (SpMV), sparse matrix-sparse vector multiplication (SpMSpV), sparse tensor-times-vector (TTV), sparse tensor-times-sparse vector (SpTSpV), and sparse matricized-tensor times Khatri-Rao product (MTTKRP). We focus on kernels generating dense outputs since writing into sparse structures through our format algebra remains future work.

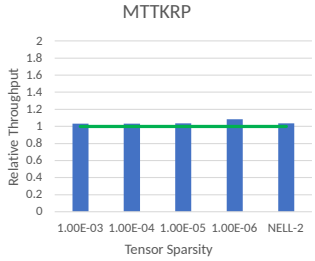
Figure 9 reports the throughput of ATL-generated programs relative to TACO on the same formats and input tensors. The green line represents the performance of TACO, or a relative throughput (speedup) of 1. We ran all experiments as single-threaded programs on a 2.8GHz Intel Core i5-8400 machine with 9 MB of L3 cache and 24 GB of main memory. All programs were compiled using `clang 17` with `-O3` and `-ffast-math` flags enabled.

9.1 Sparse Matrix Computations

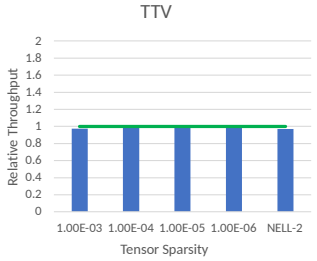
We evaluate sparse matrix–vector multiplication (SpMV) to showcase the breadth of our framework's sparse-matrix-format support, as it is both ubiquitous in sparse linear algebra and used in TACO's own evaluation. Using the sparse format abstraction and the ATL rewrite framework, we derived ATL programs with input matrices stored in block-compressed sparse row (BCSR), double-compressed sparse row (DCSR), compressed-sparse column (CSC), double-compressed sparse column (DCSC), coordinate-list (COO), and compressed sparse row (CSR) formats. We chose these formats as they are the ones readily available in the stable release build of TACO [Chou et al. 2018]. Our experiments used real-world sparse matrix inputs from the SuiteSparse Matrix Collection [Davis and Hu 2011]. These matrices ranged in size from 10^8 to 10^{12} total elements (including zeros) and nonzero densities from 3×10^{-6} to 4×10^{-3}. Figure 9a contains the evaluation results for each format in a separate graph. Overall, we found that ATL-generated code is comparable to TACO's performance across formats, except for COO and BCSR. We analyze both cases below.



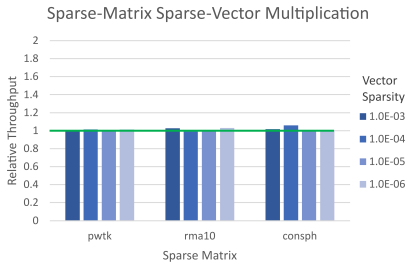
(a) Sparse matrix-vector multiplication.



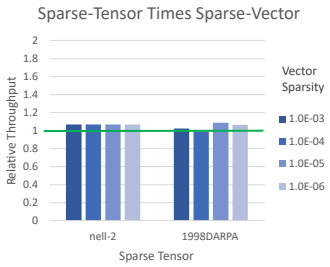
(b) Matricized tensor times Khatri-Rao product.



(c) Tensor times vector.



(d) Sparse matrix-sparse vector multiplication.



(e) Sparse tensor-sparse vector multiplication.

Fig. 9. The relative speedup of ATL-generated kernels against TACO-generated kernels of the same formats

9.1.1 COO Results. Figure 9a shows for various input matrices, ATL-generated code for COO has worse throughput than TACO with a geometric-mean relative throughput across all workloads of 0.78. The reason is likely an additional optimization performed by the TACO compiler. TACO recognizes that the coordinate arrays in COO are ordered and non-unique. Therefore, when TACO is reducing into coordinates from this array, it batches a set of reductions to one output index

by searching for the end of the segment containing the same index. The effectiveness of this optimization varies depending on the sparsity pattern of the matrix. The overhead of the loop searching for the end of a segment is outweighed only if there were a significant number of nonzero elements with that given coordinate. This difference explains why the magnitude of the peformance gap between TACO and ATL varies. Currently, this optimization is not expressible in ATL, but we acknowledge it as valuable future work.

9.1.2 BCSR Results. The ATL-generated code for BCSR is able to outperform TACO with a geometric mean of relative throughput across all inputs of 1.6. In TACO, generating a BCSR kernel requires manually constructing a 4-D tensor format, so the compiler generates inner loops whose extents correspond to input tensor dimensions. It lacks knowledge that these inner dimensions are the tiling factors. In contrast, a programmer using the ATL framework begins from a 2-D tensor and introduces tiling through rewrites, making the tile size explicit at compile time. Consequently, ATL emits loops with constant extents, which downstream compilers like `clang` can better optimize.

9.2 Sparse Tensor Computations

We chose TTV and MTTKRP as the focus of our higher-order tensor evaluation because they are widely used building blocks in sparse tensor computations. TTV is a core operation in many tensor-decomposition algorithms, while MTTKRP is the computational bottleneck in canonical polyadic (CP) decomposition, a common tensor-factorization algorithm. Figures 9b and 9c report results for TTV and MTTKRP kernels, shown in separate graphs. All input tensors were third-order tensors stored in CSF format. We ran these kernels on a range of synthetic tensor sparsities from 10^{-3} to 10^{-6} with sizes of around 10^9 total elements and the real-world sparse tensor NELL-2 from the FROSTT tensor collection [Smith et al. 2017], which has a size of about 3×10^{12} total elements and a nonzero density of approximately 2×10^{-5}. Across both kernels and all inputs, ATL-generated kernels consistently matched the performance of TACO equivalents.

9.3 Computations with Multiple Sparse Operands

We also derived a sparse-matrix sparse-vector multiplication kernel (Figure 9d) and a sparse-tensor times sparse-vector kernel (Figure 9e), to demonstrate the ability of our framework to support multiple sparse operands. For these benchmarks, we chose a subset of the real-world sparse matrices and sparse tensors from the single-sparse-operand tests that had the largest dimension sizes and most nonzeros. The matrices were stored in CSR and the tensors in DCSR and multiplied against synthesized sparse vectors stored in compressed format at a range of sparsities ranging from 10^{-3} to 10^{-6}. Across both kernels and all inputs, ATL-generated kernels consistently matched the performance of TACO equivalents.

10 Related Work

Work on sparse matrix and tensor computations has a long and rich history. A variant of the CSR format was described in early work by Tinney and Walker [1967]. Subsequently, many sparse matrix formats have been proposed to optimize both for different types of sparse matrices and for execution on different hardware, such as ELL and ELLPACK [Kincaid et al. 1989] and the HYB format [Guo et al. 2016]. Blocked-CSR format enables vectorization and reduces indexing overhead, making it suitable for high-performance execution on modern CPUs [Jain 2009; Vuduc et al. 2005]. Many libraries support a subset of these sparse formats, including cuSparse [NVIDIA 2021], MKL [Intel 2025], and SciPy's sparse package [SciPy 2025].

Researchers have also explored compilers for sparse tensor algebra. Early works include Bernoulli [Kotlyar et al. 1997] and Bik and Wijshoff [1996], both of which, like our work, use dense specifications to derive equivalent sparse programs. More recently, the TACO compiler [Kjolstad et al. 2017] takes as input expressions in a variant of Einstein notation as well as formats per-operand, generating sparse code adapted to the specific set of formats.

Our approach to levelized format definitions is inspired by Chou et al. [2018], who extended the format abstraction in TACO. Our work differs in that our abstraction is built to allow a high-level source-to-source encoding of and soundness description of abstract compression routines and their formats rather than perform code generation like the TACO formats work. As a result, our format abstraction contains a more-minimal function interface, and it concretizes the soundness properties required by level-format instances to provide for these functions in order for the compression semantics to be sound.

The work of Kovach et al. [2023] formalizes a lowering of TACO code to a correct-by-construction stream-based IR. Also, the works of Kjolstad et al. [2019] and Senanayake et al. [2020] propose an IR for sparse tensor algebra with loops that are also optimized via rewrites. Root et al. [2024] present a compiler that enables sparse tensor reshaping similar to dense reshapes supported by NumPy. Looplets [Ahrens et al. 2023] is a language for generating loops that coiterate over different sparse structures, necessary for computations with multiple sparse operands or sparse outputs. Kumar et al. [2017] builds a framework that allows specifying array computations in a functional manner, while achieving the same asymptotic complexity as imperative implementations. Liu et al. [2024b] present an MLIR-based system that allows customization of sparse formats beyond those supported by level-attribute format specifications such as those used by TACO, enabling even further specialization of formats for specific hardware and data properties.

More recently, researchers have been building tools and compilers that leverage formal methods to ensure the correctness of generated code. Bik and Wijshoff [1996] uses conditional guards and attribute grammars to determine if guards can be removed in the sparse implementation, but the tranformations remain unverified. Gladshtein et al. [2024] builds a separation logic for reasoning about loops computing over compressed data and a set of automation techniques embedded in Rocq that makes such proofs compact and easier to write. Schleich et al. [2023] and Ghorbani et al. [2023] build frameworks to specify the computation and storage structure of tensors and automatically generate code adapted to these structures. In particular, Schleich et al. [2023] and the follow-up work by Shaikhha et al. [2024] allow users to define decompressions as storage mappings and compose them with tensor programs before optimizing using equality saturation and cost models. Our system also supports introducing sparse structures into tensor programs with user-extensible formats and optimizing the resulting program. However, our format abstraction is parametric and dimensionally compositional. Rather than writing a new decompression procedure for each format, users compose reusable, proven level-format instances, producing correct-by-construction. These works use declarative formalisms to specify how code is generated but do not provide formal verification of the generated code. While the ATL framework is currently user-scheduled, the rewrite system is amenable to optimization via equality saturation and cost modeling.

11 Conclusion

We present the first mechanized, formally verified algebraic framework for compositionally constructing sparse tensor formats. We implemented it as an extension of the ATL scheduling framework, which had previously only supported dense tensor operands [Liu et al. 2024a, 2022]. Our extension includes a formalized abstraction for levelized encoding schemes on tensors that are used to build multidimensional sparse tensor formats and format-agnostic soundness theorems that allow us to generate sparse kernel code, derived to be semantically equivalent to an original

dense tensor program. We show that a more-minimal set of core level formats than previous works [Chou et al. 2018], combined with the existing concept of reshape operators from ATL, is sufficient to recover numerous common tensor compression formats. Overall, we demonstrate that compiling the derived sparse ATL programs using a minimally extended version of the existing ATL lowering algorithm yields performance competitive with TACO, a state-of-the-art sparse tensor compiler.

12 Acknowledgements

This work was done in part under the support of an internship at Adobe Research. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-2313023 and CCF-2217064, in addition to the Defense Advanced Research Projects Agency (DARPA) contract HR001125CE038 (MOCHA program). We also thank Manya Bansal for her help in understanding and navigating the TACO compiler codebase, which was instrumental in benchmarking our proposed approach.

References

Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 41–54. doi:10.1145/3579990.3580020

A.J.C. Bik and H.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. doi:10.1109/71.485501

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. doi:10.1145/3276493

Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. doi:10.1145/2049662.2049663

Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 229 (Oct. 2023), 30 pages. doi:10.1145/3622804

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI, Article 173 (June 2024), 24 pages. doi:10.1145/3656403

Dahai Guo, William Gropp, and Luke N Olson. 2016. A hybrid format for better performance of sparse matrix-vector multiplication on a GPU. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 103–120. doi:10.1177/1094342015593156

Intel. 2025. Intel Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html

Kenneth E. Iverson. 1962. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA. doi:10.5555/1098666

Ankit Jain. 2009. *pOSKI: An extensible autotuning framework to perform optimized SpMVs on multicore architectures*. Master's thesis. University of California, Berkeley. https://bebop.cs.berkeley.edu/pubs/jain2008-masters.pdf

David R. Kincaid, Thomas C. Oppe, and David M. Young. 1989. ITPACKV 2D user's guide. https://api.semanticscholar.org/CorpusID:56590066

Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor algebra compilation with workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 180–192. doi:10.1109/CGO.2019.8661185

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. doi:10.1145/3133901

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A Relational Approach to the Compilation of Sparse Matrix Programs. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*. Springer-Verlag, London, UK, 318–327. http://iss.ices.utexas.edu/Publications/Papers/EUROPAR1997.pdf

Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. 2023. Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 154 (jun 2023), 25 pages. doi:10.1145/3591268

Ananya Kumar, Guy E. Blelloch, and Robert Harper. 2017. Parallel functional arrays. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 706–718. doi:10.1145/3009837.3009869

Amanda Liu. 2026. *Artifact for A Mechanized Algebra of Verified Data Structures for Optimizing Sparse Tensor Programs*. doi:10.5281/zenodo.19422499

Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2024a. A Verified Compiler for a Functional Tensor Language. *Proc. ACM Program. Lang.* 8, PLDI, Article 160 (June 2024), 23 pages. doi:10.1145/3656390

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (Jan. 2022), 28 pages. doi:10.1145/3498717

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

NVIDIA. 2021. cuSPARSE. https://docs.nvidia.com/cuda

Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. 2024. Compilation of Shape Operators on Sparse Arrays. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 312 (Oct. 2024), 27 pages. doi:10.1145/3689752

Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1, Article 37 (May 2023), 27 pages. doi:10.1145/3588717

SciPy. 2025. SciPy. https://www.scipy.org/

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. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. doi:10.1145/3428226

Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. 2024. A Tensor Algebra Compiler for Sparse Differentiation. *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2024), 1–12. https://api.semanticscholar.org/CorpusID:268045813

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/

W.F. Tinney and J.W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809. doi:10.1109/PROC.1967.6011

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, 1 (Jan. 2005), 521. doi:10.1088/1742-6596/16/1/071

Liang Yuan, Yunquan Zhang, Xiangzheng Sun, and Ting Wang. 2010. Optimizing Sparse Matrix Vector Multiplication Using Diagonal Storage Matrix Format. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*. 585–590. doi:10.1109/HPCC.2010.67

Received 2025-11-12; accepted 2026-04-03