Automatic Generation of Efficient Sparse Tensor Format Conversion Routines

Stephen Chou
MIT CSAIL
Cambridge, MA, USA
s3chou@csail.mit.edu

Fredrik Kjolstad
Stanford University
Stanford, CA, USA
kjolstad@cs.stanford.edu

Saman Amarasinghe
MIT CSAIL
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

This paper shows how to generate code that efficiently converts sparse tensors between disparate storage formats (data layouts) such as CSR, DIA, ELL, and many others. We decompose sparse tensor conversion into three logical phases: coordinate remapping, analysis, and assembly. We then develop a language that precisely describes how different formats group together and order a tensor's nonzeros in memory. This lets a compiler emit code that performs complex remappings of nonzeros when converting between formats. We also develop a query language that can extract statistics about sparse tensors, and we show how to emit efficient analysis code that computes such queries. Finally, we define an abstract interface that captures how data structures for storing a tensor can be efficiently assembled given specific statistics about the tensor. Disparate formats can implement this common interface, thus letting a compiler emit optimized sparse tensor conversion code for arbitrary combinations of many formats without hard-coding for any specific combination.

Our evaluation shows that the technique generates sparse tensor conversion routines with performance between 1.00 and 2.01× that of hand-optimized versions in SPARSKIT and Intel MKL, two popular sparse linear algebra libraries. And by emitting code that avoids materializing temporaries, which both libraries need for many combinations of source and target formats, our technique outperforms those libraries by 1.78 to 4.01× for CSC/COO to DIA/ELL conversion.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; **Source code generation**; *Domain specific languages*; • **Mathematics of computing** → *Mathematical software performance.*

Keywords: sparse tensor conversion, sparse tensor assembly, sparse tensor algebra, sparse tensor formats, coordinate remapping notation, attribute query language

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385963

ACM Reference Format:
Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385963

1 Introduction

Sparse multidimensional arrays (tensors) are suited for representing data in many domains, including data analytics [2, 6], machine learning [41, 46], and others. Countless formats for storing sparse tensors have been developed [5, 7, 8, 10, 13, 14, 23, 27, 34–37, 47, 49, 50, 53, 55, 60, 61] to accelerate kernels like sparse matrix-vector multiplication (SpMV), and new formats are constantly being proposed in recent literature.

No format is universally superior in every circumstance, since the ideal format for storing a sparse tensor depends on its structure and sparsity, the operation being performed, and the available hardware. Applications typically need to perform different operations on the same tensor, and each operation may require the tensor to be stored in a distinct format for optimal performance. Importing data into a sparse tensor, for instance, can be done efficiently if the tensor is constructed in the COO format [7] or the DOK format [54], since they support efficient appends or random insertions of new nonzeros. Computing SpMV with the tensor, however, can be done more than twice as fast if the tensor is stored in CSR [55], which compresses out redundant row coordinates and thereby reduces memory traffic [17]. Alternatively, if all of the tensor's nonzeros are clustered along a few dense diagonals, then storing it in DIA [49] minimizes memory traffic even more while exposing vectorization opportunities, further improving SpMV performance by up to 22% as a result [17]. Thus, to optimize the performance of both data import and compute, an application must convert the tensor from COO (or DOK) to DIA (or CSR). And in applications like preconditioned solvers and sparse neural network training where a tensor might only be computed with a few times, the conversion must be efficient so that the overhead does not outweigh gains from using an optimized format [20].

General-purpose sparse linear and tensor algebra libraries like SPARSKIT [48] and Intel MKL [24] thus strive to support efficiently converting tensors between as many formats as

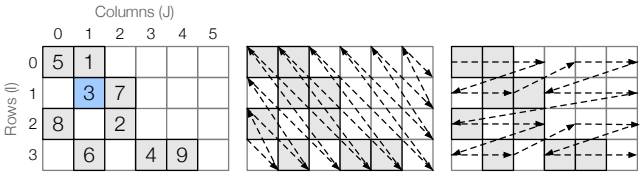


Figure 1. A sparse 4×6 tensor (matrix). Nonzeros can be ordered in memory not only by rows or by columns, but also by diagonals (center) or even by blocks (right).

possible. SPARSKIT, for instance, supports no fewer than 17 different matrix formats. With such large numbers (N) of supported formats, it becomes impractical to manually implement efficient conversion routines for all $\Theta(N^2)$ combinations of source and target formats. Instead, hand-optimized libraries typically only support direct conversions to and from some arbitrary canonical format (e.g., CSR with SPARSKIT). Thus, to convert a tensor from COO to DIA using SPARSKIT (or Intel MKL), an application must first convert the tensor to CSR and then to DIA. This doubles the number of conversions needed, which is inefficient when converting a tensor even once incurs significant overhead [17, 60]. Worse, this approach is not even feasible if an application uses novel formats that are not supported by libraries, such as many of the sparse matrix and tensor formats that have been proposed in recent literature. The developer must then hand-implement efficient custom conversion routines for each new format, which are typically complicated and tedious to write and optimize. This motivates a technique that can instead automatically generate such efficient format conversion routines.

Existing sparse tensor algebra compilers such as taco [17, 29, 30] are unable to generate such routines for many formats. Converting a tensor between disparate formats typically entails changing how its nonzeros are grouped and ordered in memory, potentially rearranging nonzeros into complex orders such as by diagonals or by blocks (Figure 1), or even by Morton order [13, 34]. Efficient conversion algorithms can often achieve this reordering without explicitly sorting the input tensor by first computing statistics about the tensor. These statistics are then used to coordinate the movement of nonzeros to the output tensor (in the target format) in such a way that avoids data reshuffles and memory reallocations. The taco compiler cannot generate such algorithms as it cannot express or reason about reordering nonzeros in non-lexicographic coordinate order. It also cannot reason about computing and utilizing statistics about the input tensor to coordinate assembly of disparate tensor data structures.

We propose a technique to generate efficient sparse tensor conversion routines for a wide range of disparate formats, building on our recent works on sparse tensor algebra compilation [17, 29, 30]. We decompose a large class of tensor conversion algorithms into three logical phases (Section 3). Then, to facilitate generating code for each phase, we develop

coordinate remapping notation, which describes how different tensor formats group together and order nonzeros in memory (Section 4);

attribute query language, which describes what statistics about a tensor are needed so that sufficient memory can be reserved for conversion (Section 5); and a

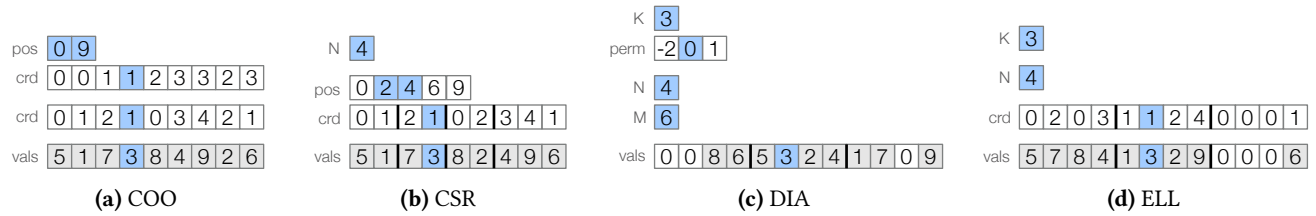
tensor assembly abstract interface, which exposes functions that capture how results of attribute queries are used to efficiently assemble many kinds of sparse tensor data structures (Section 6).

As we will show, the conciseness of these abstractions makes it easy to provide specifications that describe how to efficiently construct sparse tensors in many formats. Our technique can then combine these specifications with additional ones proposed by Chou et al. [17], which specify how to efficiently iterate over sparse tensors in many formats, in order to generate efficient conversion routines for arbitrary combinations of formats. In this way, users only have to provide one set of specifications for every supported format rather than every combination of source and target formats.

We have implemented a prototype of our technique in taco. Our evaluation shows that, for many combinations of source and target formats, our technique generates conversion routines with performance between 1.00 and 2.01× that of hand-optimized implementations in SPARSKIT and Intel MKL. For conversions from CSC/COO to DIA/ELL, our technique emits code that avoid materializing temporaries and that are not implemented in either library, which lets us optimize those conversions by 1.78 to 4.01× (Section 7).

2 Background

There exist a wide variety of formats for storing sparse tensors in memory. Figure 2 shows four examples of commonly used sparse tensor formats; for an overview of more formats that have been proposed, we refer readers to Section 2.1 in [17]. The COO format [7] represents a sparse tensor as a list of its nonzeros, storing the complete coordinates and value of each nonzero. COO supports efficiently appending new nonzeros, though it also wastes memory by storing redundant row coordinates. The CSR format [55] compresses out the redundant row coordinates by grouping all nonzeros in the same row together and using a pos array to map nonzeros to each row. However, inserting a nonzero at some arbitrary coordinates into CSR is expensive as all nonzeros in subsequent rows must be shifted in memory. The DIA [49] format stores nonzeros along the same diagonal together in memory, while the ELL [27] format groups together up to one nonzero from each row. Such orderings of nonzeros expose vectorization opportunities for SpMV [19] and can also reduce memory footprint. However, DIA is only suitable for diagonal and banded matrices, while ELL is only suitable if all rows in the matrix have a similar number (K) of nonzeros. As these examples show, there is no universally ideal format



(a) COO **(b)** CSR **(c)** DIA **(d)** ELL

Figure 2. The same tensor as shown in Figure 1, stored in disparate sparse tensor formats.

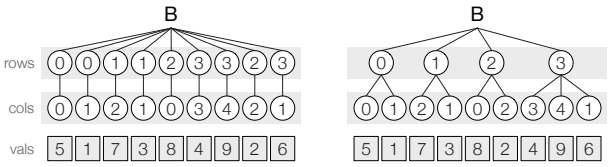


Figure 3. Two coordinate hierarchy representations of the same tensor from Figure 1 in COO (left) and CSR (right). Their differing structures reflect how COO and CSR store nonzeros (i.e., whether duplicate row coordinates are stored).

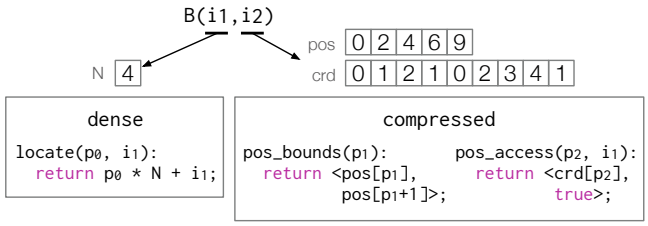


Figure 4. Decomposition of CSR into level formats and corresponding level functions that describe how the associated data structures can be efficiently accessed.

for storing sparse matrices. The same is true of higher-order sparse tensors, for which even more formats have been—and are continuously being—proposed [8, 34, 35, 50] that use disparate data structures and ordering schemes to store nonzeros, each with distinct trade-offs.

Chou et al. [17] describe how tensors stored in disparate formats can be represented as *coordinate hierarchies* that have varying structures but that expose the same abstract interface. Figure 3 shows examples of coordinate hierarchies that represent a tensor in two different formats. Each level in a coordinate hierarchy encodes the nonzeros' coordinates into one dimension. Edges associate stored components with their containing subtensors. In Figure 3, for instance, each column coordinate, which is associated with a nonzero, is connected by an edge to a coordinate that identifies the nonzero's containing row. Each stored component is represented by a path from the root to a leaf, with coordinates along the path representing the component's coordinates. We refer readers to Section 3 in [17] for more details.

We can then decompose sparse tensor formats into *level formats* that each stores a coordinate hierarchy level, which represents a tensor dimension. CSR (Figure 2b), for instance, can be decomposed into two level formats, *dense* and *compressed*, that store the row and column levels respectively, as Figure 4 shows. The dense level format implicitly encodes all rows using just a parameter `N` to store the dimension's size. By contrast, the compressed level format uses two arrays, `pos` and `crd`, to store column coordinates of nonzeros in each row. All level formats, however, expose the same static interface consisting of *level functions*, which describe how to access a format's data structures, and *properties*, which describe characteristics of the data as stored (e.g., if nonzeros are stored in order). The level function `locate`, for instance, describes how to efficiently random access coordinates that are stored in a level format. Similarly, `pos_bounds` and `pos_access` describe how to efficiently iterate over coordinates, with the former specifying how to compute the bounds of iteration and the latter specifying how to access each coordinate.

Structured sparse tensor formats like DIA and ELL, which do not group nonzeros lexicographically by their coordinates, can also be decomposed into level formats by casting them as formats for tensors with additional dimensions. For example, a DIA matrix can be cast as a 3rd-order tensor where each slice contains only nonzeros that lie on the same diagonal, as shown in Figure 5. We can then decompose DIA into three level formats: one that stores the set of nonzero diagonals in a `perm` array of size `K`, another that encodes the set of rows in each diagonal, and a third that encodes the column coordinates of nonzeros. Such a decomposition lets a sparse tensor algebra compiler reason about how tensors stored in DIA and similar structured formats can be efficiently iterated, which is crucial for generating fast tensor algebra code.

The coordinate hierarchy abstraction lets a compiler generate efficient code to iterate over sparse tensors in disparate formats by simply emitting code to traverse coordinate hierarchies. This entails recursively generating a set of nested loops that each iterates over a level in a coordinate hierarchy. The compiler generates each loop by emitting calls to level functions that describe how to efficiently access the level. Then, to obtain code that iterates over a tensor in any desired format, the level function calls are simply replaced with the desired format's implementations of those level functions. This approach lets a compiler generate efficient code for disparate formats without hard-coding for any specific format. We refer readers to Section 4.5 in [17] for more details.

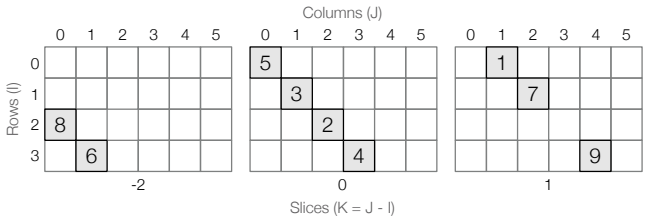


Figure 5. The matrix in Figure 1 can be transformed to a 3rd-order tensor where each slice contains all nonzeros that lie on the same diagonal in the original matrix. The lexicographic coordinate ordering of nonzeros in the resulting tensor matches the order in which nonzeros are stored in DIA (Figure 2c). Section 4 shows how this transformation is formalized by the coordinate remapping `(i,j) -> (j-i,i,j)`.

3 Overview

Figure 6 shows three examples of sparse tensor conversion routines that efficiently convert tensors between different storage formats. As these examples illustrate, different combinations of source and target formats require vastly dissimilar code. It turns out, however, that efficient algorithms for converting sparse tensors between a wide range of disparate formats can all be decomposed into three logical phases: coordinate remapping, analysis, and assembly. Figure 6 identifies these phases using different background colors.

The coordinate remapping phase iterates over the input tensor and, for each nonzero, computes new coordinates as functions of its original coordinates. What additional coordinates are computed depends on the target format. For instance, the code in Figure 6a, which converts a CSR matrix to DIA, computes a new coordinate `k` for each nonzero as the difference between its column and row coordinates (lines 2–6 and 20–24). Coordinate remapping conceptually transforms (i.e., remaps) the input tensor to a hypersparse higher-order tensor. As Figure 5 illustrates, the lexicographic coordinate ordering of nonzeros in the remapped tensor reflects how nonzeros are grouped together and ordered in the target format (i.e., DIA), even if the format does not store nonzeros in lexicographic order by their original coordinates.

The analysis phase computes statistics about the input tensor that are later used to determine the amount of memory to allocate for storing nonzeros in the target format. The exact statistics that are computed also depend on the target format. Figure 6a, for instance, computes the set of all nonzero diagonals in the input matrix (lines 1–8), with distinct diagonals identified by offsets (`k`) computed in the coordinate remapping phase. By contrast, Figure 6b computes the maximum number of nonzeros in any row of the input matrix (lines 1–5), while Figure 6c computes the number of nonzeros in each row of the input matrix (lines 1–6).

Finally, the assembly phase iterates over the input tensor and inserts each nonzero into the output data structures. Again, where each nonzero is inserted (`pB2`) depends on the target format. Figure 6a computes `pB2` as a function of each nonzero's row coordinate and its offset `k` (as computed in the coordinate remapping phase), in such a way that nonzeros with the same offset are grouped together in the output (lines 25–26). By contrast, Figure 6c simply appends each nonzero to its row's corresponding segment in the `crd` array (line 19).

In Sections 4 through 6, we describe how our technique generates efficient code to perform coordinate remapping (Section 4), analysis (Section 5), and assembly (Section 6). For each logical phase, we define a language that captures what needs to be performed for disparate target formats. Figure 7 demonstrates how these languages can be used to specify what needs to be performed when converting tensors to ELL. For each new target tensor format, a user must first specify

- a coordinate remapping (in green) that, when applied to the input tensor, captures how nonzeros are grouped together and ordered in the target format.

As alluded to in Section 2, the target tensor format can then be decomposed into level formats that each stores a dimension of the remapped input tensor. For each of these level formats, the user must then also specify

- what input tensor statistics to compute (in yellow) and
- how to store the coordinates of nonzeros (in blue).

To generate code that converts tensors between any two specific formats, our technique combines the aforementioned specifications for the target format with level functions that describe how to iterate over tensors in the source format. Given the specifications in Figures 4 and 7 as inputs, for instance, our technique generates code like what is shown in Figure 6b, which performs CSR to ELL conversion. Just as easily though, given the same specifications in Figure 7 but also level functions that describe how to iterate over COO tensors, our technique instead generates efficient COO to ELL conversion code. In this way, our technique can generate efficient conversion routines for many combinations of formats without needing specifications for each combination.

Having a separate language to describe each logical phase provides several benefits. First, converting to different tensor formats may require similar steps to be taken for only some of the phases, so having each phase be specified separately allows for reuse of the specifications. For instance, the COO format uses the same data structure as ELL to store column coordinates. Thus, the two formats can share specifications for assembly (i.e., level functions implemented for the singleton level format) even if they require different coordinate remappings. Second, having each logical phase be specified separately gives the compiler flexibility to generate code that fuses logically distinct phases only if it is beneficial. Our technique can thus generate code like Figure 6a, which duplicates and fuses coordinate remapping with the analysis and assembly phases to avoid materializing the offsets of nonzeros. At the same time, for conversions to formats that store nonzeros in more complex orderings (e.g., Morton order), the

```
bool nz[2 * N - 1] = {0};
for (int i = 0; i < N; i++) {
  for (int pA2 = A_pos[i];
           pA2 < A_pos[i+1]; pA2++) {
    int j = A_crd[pA2];
    int k = j - i;
    nz[k + N - 1] = true;
}}
int* B_perm = new int[2 * N - 1];
int K = 0;
for (int i = -N + 1; i < N; i++) {
  if (nz[i + N - 1])
    B_perm[K++] = i;
}
double* B_vals = new double[K * N]();
int* B_rperm = new int[2 * N - 1];
for (int i = 0; i < K; i++) {
  B_rperm[B_perm[i] + N - 1] = i;
}
for (int i = 0; i < N; i++) {
  for (int pA2 = A_pos[i];
           pA2 < A_pos[i+1]; pA2++) {
    int j = A_crd[pA2];
    int k = j - i;
    int pB1 = B_rperm[k + N - 1];
    int pB2 = pB1 * N + i;
    B_vals[pB2] = A_vals[pA2];
}}
```

(a) CSR to DIA

```
int K = 0;
for (int i = 0; i < N; i++) {
  int ncols = A_pos[i+1] - A_pos[i];
  K = max(K, ncols);
}
int* B_crd = new int[K * N]();
double* B_vals = new double[K * N]();
for (int i = 0; i < N; i++) {
  int count = 0;
  for (int pA2 = A_pos[i];
           pA2 < A_pos[i+1]; pA2++) {
    int j = A_crd[pA2];
    int k = count++;
    int pB2 = k * N + i;
    B_crd[pB2] = j;
    B_vals[pB2] = A_vals[pA2];
}}
```

(b) CSR to ELL

```
int count[N] = {0};
for (int pA1 = A_pos[0];
         pA1 < A_pos[1]; pA1++) {
  int i = A1_crd[pA1];
  count[i]++;
}
int* B_pos = new int[N + 1];
B_pos[0] = 0;
for (int i = 0; i < N; i++) {
  B_pos[i + 1] = B_pos[i] + count[i];
}
int* B_crd = new int[pos[N]];
double* B_vals = new double[pos[N]];
for (int pA1 = A_pos[0];
         pA1 < A_pos[1]; pA1++) {
  int i = A1_crd[pA1];
  int j = A2_crd[pA1];
  int pB2 = pos[i]++;
  B_crd[pB2] = j;
  B_vals[pB2] = A_vals[pA2];
}
for (int i = 0; i < N; i++) {
  B_pos[N - i] = B_pos[N - i - 1];
}
B_pos[0] = 0;
```

(c) COO to CSR

Figure 6. Code (in C++) to convert sparse tensors between different source and target formats. The background colors identify distinct logical phases of format conversion (green for coordinate remapping, yellow for analysis, and blue for assembly).

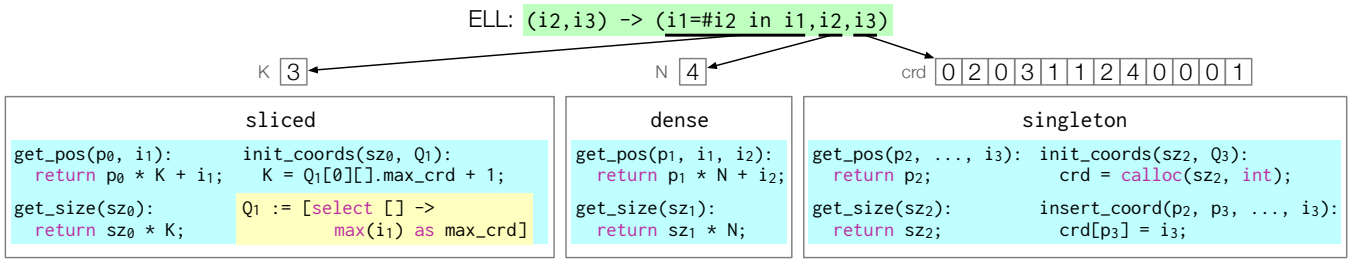


Figure 7. Specifications that describe what needs to be performed as part of each logical phase when converting sparse tensors to ELL, which can be decomposed into three level formats: *sliced*, *dense*, and *singleton*. The background colors identify the phases being specified, following the same scheme as Figure 6.

compiler can emit code to perform coordinate remapping separately and materialize the additional coordinates. This eliminates the need to recompute complex remappings.

4 Coordinate Remapping

As explained in Section 3, many efficient sparse tensor conversion algorithms logically transform (i.e., remap) input tensors to higher-order tensors, such that the lexicographic coordinate ordering of nonzeros in the remapped tensors specify how nonzeros are stored in the target formats. We propose a new language called *coordinate remapping notation*, which precisely describes how a tensor can be remapped so as to capture the various ways that different tensor formats group together and order nonzeros in memory. We further show how our technique generates code that applies a coordinate remapping to remap the input tensor as part of format conversion. This eliminates the need for end users to hand-implement additional code that separately performs such a remapping, which the technique of Chou et al. [17] requires for conversions to structured tensor formats.

4.1 Coordinate Remapping Notation

Figure 8 shows the syntax of coordinate remapping notation. Statements in coordinate remapping notation specify how components in a canonical (non-remapped) input tensor map to components in an output tensor of equal or higher order. For instance, given a matrix A as input, the statement

```
(i,j) -> (j-i,i,j)
```

maps every component A_{ij} to the corresponding component in the $(j-i)$-th slice of three-dimensional remapped tensor. Applying this remapping to any matrix, which can be stored in any format but must have canonical coordinates

```
remap_stmt := src_indices '->' dst_indices
src_indices := '(' ivar { ',' ivar } ')'
dst_indices := '(' ivar_let { ',' ivar_let } ')'
ivar_let := { var '=' ivar_expr 'in' } ivar_expr
ivar_expr := ivar_xor { '|' ivar_xor }
ivar_xor := ivar_and { '^' ivar_and }
ivar_and := ivar_shift { '&' ivar_shift }
ivar_shift := ivar_add { ( '<<' | '>>' ) ivar_add }
ivar_add := ivar_mul { ( '+' | '-' ) ivar_mul }
ivar_mul := ivar_factor { ( '*' | '/' | '%' ) ivar_factor }
ivar_factor := '(' ivar_expr ')' | ivar_counter | ivar | var | const
ivar_counter := '#' { ivar }
```

Figure 8. The syntax of coordinate remapping notation. Expressions in braces may be repeated any number of times.

(i, j), transforms it to a 3rd-order tensor where each slice contains all nonzeros that lie on the same diagonal in the original matrix. As Figure 5 shows, the lexicographic coordinate ordering of nonzeros in the resulting tensor precisely reflects the order in which nonzeros are stored in DIA.

Similarly, the BCSR format partitions a matrix into fixed-sized $M \times N$ blocks and stores components of each block contiguously in memory [23]. Such grouping of nonzeros can be expressed with the remapping

```
(i,j) -> (i/M,j/N,i,j),
```

which assigns components that lie within the same block to the same two-dimensional slice (identified by coordinates $(i/M, j/N)$) in the output tensor.

Coordinate remapping notation can express complex tensor reorderings. The remapping below, for instance, groups together nonzeros that lie within the same constant-sized $N \times N \times N$ block and also orders the blocks as well as the nonzeros within each block in Morton order [38]:

```
(i,j,k) ->
  (r=i/N in s=j/N in t=k/N in
     (r&1) | ((s&1)<<1) | ((t&1)<<2) ...,i/N,j/N,k/N,
   u=i%N in v=j%N in w=k%N in
     (u&1) | ((v&1)<<1) | ((w&1)<<2) ...,i,j,k).
```

This remapping exactly describes how the HiCOO tensor format orders nonzeros in memory [34]. Nested let expressions are used to first define variables `r`, `s`, and `t` as the coordinates of each block and variables `u`, `v`, and `w` as the coordinates of each nonzero within a block. The remapping then computes the Morton code of each block and each nonzero within a block by interleaving the bits of those previously defined coordinates using bitwise operations.

Coordinate remapping notation also provides counters, denoted by "`#`" in Figure 8. Counters map nonzeros that share the same specified coordinates to distinct slices in the result. For instance, as Figure 9 shows, the remapping

```
(i,j) -> (k=#i in k,i,j)
```

assigns the k-th nonzero that is iterated over in each row of the input matrix to the k-th slice in the output tensor,

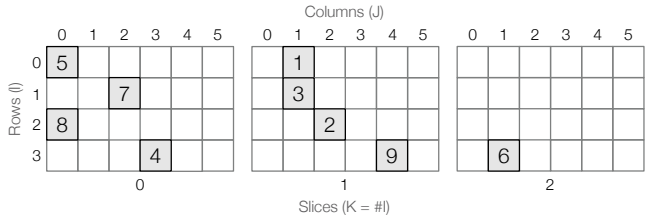


Figure 9. Result of applying `(i,j) -> (#i,i,j)` to the matrix in Figure 1, assuming nonzeros are iterated over during remapping in the same order as they are stored in Figure 2b.

ensuring nonzeros with the same i coordinate are mapped to distinct slices. This remapping effectively groups together up to one nonzero from each row of the input matrix, accurately reflecting how formats like ELL and JAD [47] store nonzeros.

4.2 Code Generation

To support generating sparse tensor conversion routines for arbitrary combinations of formats, we independently annotate each supported format with a coordinate remapping that describes how the format groups together and orders nonzeros. As the previous examples demonstrate, it is also simple for end users to add support for additional custom formats by using coordinate remapping notation to specify how the new formats lay out nonzeros in memory. Then, to support converting tensors between two specific formats, our technique emits code that iterates over the input tensor and transforms the (canonical) coordinates of each nonzero by applying the target format's coordinate remapping.

To generate a set of nested loops that efficiently iterate over an input tensor in any source format, our technique uses the method proposed by Kjolstad et al. [30] and generalized by Chou et al. [17], which is summarized at the end of Section 2. Within the generated loops, our technique then emits code that computes each remapped nonzero's additional coordinates as functions of the nonzero's original coordinates, following the target format's coordinate remapping.

To compute additional coordinates that are defined purely as arithmetic or bitwise expressions of the original coordinates, our technique simply inlines those expressions directly into the emitted code (e.g., lines 6 and 24 in Figure 6a, which compute the first coordinate in the output of the remapping `(i,j) -> (j-i,i,j)`). Remappings that contain let expressions are lowered by first emitting code to initialize the local variables and then inlining the expressions that use those local variables. For example, a remapped coordinate `r=i/N in (r&1) | ((r&2)<<2)` would be lowered to

```
int r = i/N;
int m = (r&1) | ((r&2)<<2;
```

Coordinate remappings that contain counters are lowered by emitting a counter array for each distinct counter in the remapping. Each element in the counter array corresponds to a distinct set of coordinates $(i_1, ..., i_k)$ that can be used to

index into the counter, and the counter array element tracks how many input nonzeros with coordinates $(i_1, ..., i_k)$ have been iterated over so far. Our technique additionally emits code that, for each nonzero having coordinates that correspond to counter array element c, first assigns the nonzero to the output tensor slice indexed by c and then increments c. So to apply the remapping `(i,j) -> (#i,i,j)` to a COO matrix, for instance, our technique emits the following code:

```
int counter[N] = {0}; // counter array for #i
for (int p = 0; p < nnz; p++) {
  int i = A1_crd[p];
  int j = A2_crd[p];
  int k = counter[i]++; // k == #i
  // map A(i,j) to coordinates (k,i,j) ...
```

If the coordinates used to index into a counter are iterated in order though, our technique reduces the size of the counter array in the generated code by having the counter array be reused across iterations. This can significantly reduce memory traffic. For instance, if the input matrix is instead stored in CSR, our technique infers from properties of the format (exposed through the static interface discussed in Section 2) that we can efficiently iterate over nonzeros row by row. Thus, to apply the same coordinate remapping as before, our technique emits optimized code as shown on lines 8–13 in Figure 6b, which uses the same scalar `count` variable to remap the nonzeros in each row of the input matrix.

5 Attribute Queries

As we also saw in Section 3, to avoid having to constantly reallocate and shuffle around stored nonzeros, many efficient tensor conversion algorithms instead allocate memory in one shot based on some statistics about the input tensor. Computing these statistics, however, requires very different code depending on how the input tensor is stored. For instance, to convert a matrix to ELL without dynamically resizing the `crd` and `vals` arrays, one must first determine the maximum number of nonzeros K stored in any row. If the input matrix is stored in COO, then computing K requires constructing a histogram that records the number of nonzeros in each row, which in turn requires examining all the nonzeros in the matrix. If the input is stored in CSR, however, then the number of nonzeros in each row can instead be directly computed from the `pos` array. Optimized code for converting CSR matrices to ELL thus does not need to make multiple passes over the input matrix's nonzeros, reducing memory traffic.

We propose a new language called the *attribute query language* that describes statistics of sparse tensors as aggregations over the coordinates of their nonzeros. The attribute query language is declarative, and attribute queries are specified independently of how the input tensor is actually stored. This lets our technique lower attribute queries to equivalent sparse tensor computations and then simply leverage prior work on sparse tensor algebra compilation [17, 29, 30] to generate optimized code for computing tensor statistics. As

```
select [i] ->
  count(j) as nir
```

i	nir
0	2
1	2
2	2
3	3

```
select [i] ->
  min(j) as minir,
  max(j) as maxir
```

i	minir	maxir
0	0	1
1	1	2
2	0	2
3	1	4

```
select [j] ->
  id() as ne
```

i	ne
0	1
1	1
2	1
3	1
4	1
5	0

Figure 10. Examples of attribute queries computed on the tensor shown in Figure 1.

we show in Section 6, our technique can thus generate efficient tensor conversion routines while only requiring users to provide simple-to-specify attribute queries for each potential target format, as opposed to complicated loop nests for every combination of source and target formats.

5.1 Attribute Query Language

The attribute query language lets users compute summaries of a tensor's sparsity structure by performing aggregations over the coordinates of the tensor's nonzeros. All queries in the attribute query language take the form

```
select [i1,...,im] ->
  <aggr1> as label1, ..., <aggrn> as labeln
```

where each $\texttt{i}_k$ denotes a coordinate into dimension I_k of some r-dimensional tensor A and $\texttt{<aggr}_k\texttt{>}$ invokes the aggregation function `count`, `max`, `min`, or `id`. The result of an attribute query is conceptually a map that, for every distinct set of coordinates $(i_1, \ldots, i_m)$, stores computed statistics about the $I_{m+1} \times \cdots \times I_r$ subtensor A' in A identified by those coordinates. Figure 10 shows examples of different attribute queries computed on the same tensor.

$\texttt{count(i}_{m+1}\texttt{,...,i}_l\texttt{)}$ computes the number of nonzero $I_{l+1} \times \cdots \times I_r$ subtensors, each of which can be identified by a distinct set of coordinates $(i_1, \ldots, i_l)$, that are contained in A'. For instance, if I, J, K represent the slice, row, and column dimensions of a 3rd-order tensor B, then the query

```
select [i] -> count(j) as nnr_in_slice
```

computes the number of nonzero rows contained in each $J \times K$ slice of B, while the query

```
select [i] -> count(j,k) as nnz_in_slice
```

computes the number of nonzeros in each $J \times K$ slice. Figure 10 (left) shows how `count` queries can be used to compute the number of nonzeros in each row of a matrix, which is required when converting it from COO to CSR for instance.

$\texttt{max(i}_{m+1}\texttt{)}$ and $\texttt{min(i}_{m+1}\texttt{)}$ compute, for each subtensor A', the largest and smallest coordinates i_{m+1} such that the i_{m+1}-th slice of A' along dimension I_{m+1} is nonzero. For instance, if `Q` is the result of the query in Figure 10 (middle), then

`Q[3].minir == 1` and `Q[3].maxir == 4` since all nonzeros in row 3 of the tensor in Figure 1 lie between columns 1 and 4.

Finally, `id` simply returns 1 if a subtensor A' contains nonzeros and 0 otherwise. So if `R` is the result of the query in Figure 10 (right), then `R[4].ne == 1` since column 4 contains a nonzero while `R[5].ne == 0` since the last column is empty.

The attribute query language can be used with coordinate remapping notation to compute even more complex attributes of structured tensors. For example, let A be a $K \times I \times J$ tensor obtained by applying the remapping `(i,j) -> (j-i,i,j)` to a matrix B. Since each slice of A along dimension K corresponds to a unique diagonal in B, computing

```
select [k] -> id() as ne
```

on A results in a bit set that encodes the set of all nonzero diagonals in B. This, as mentioned in Section 3, is precisely information that would be required if one were to convert B from CSR to DIA. Furthermore, since the coordinate of each slice of A is defined to be the offset of the corresponding diagonal in B from the main diagonal, applying the query

```
select [] -> min(k) as lb, max(k) as ub
```

to A computes the lower and upper bandwidths of matrix B.

5.2 Code Generation

Kjolstad et al. [29] introduce concrete index notation, which is a language for precisely specifying tensor computations. For instance, an operation that computes the sum of every row in a matrix A can be expressed as $\forall_i \forall_j \; x_i \mathrel{+}= A_{ij}$, where each $\forall$ specifies iteration over a dimension of A. Kjolstad et al. show how computations expressed in concrete index notation can be lowered to efficient imperative code; we refer readers to Section 5 in [28] for more details. At a high level, this is done recursively dimension by dimension in the order specified by the $\forall$s. To generate code for the previous example, for instance, a compiler would first emit a loop to iterate over all rows of A. Within that loop, the compiler would then emit a second loop to iterate over all columns of A in order to compute the sum of nonzeros in row i. Again, specializing the emitted code to operands in arbitrary formats can be done in the same way described at the end of Section 2.

To generate efficient code that computes an attribute query, our technique simply reformulates the query as sparse tensor algebra computation. The query is first lowered to a canonical form in concrete index notation, which we extend with the ability to index into results using coordinates that are computed as arbitrary functions of index variables. The canonical form of the query is subsequently optimized by applying a set of predefined transformations to simplify the computation. Finally, the optimized query in concrete index notation is compiled to imperative code by straightforwardly leveraging the techniques of Kjolstad et al. and Chou et al. as summarized above. This approach works as long as query results are stored in a format, like dense arrays, that can itself be efficiently assembled without needing attribute queries.

More precisely, let A be an $I_1 \times \cdots \times I_r$ tensor obtained by applying some remapping to a $J_1 \times \cdots \times J_n$ tensor B. Then, to compute an attribute query of the form

```
select [i1,...,im] -> id() as Q
```

on A for instance, our technique lowers the query to its canonical form in concrete index notation as

$$\forall_{j_1} \cdots \forall_{j_n} \; Q_{i_1 \cdots i_m} \mid\!= \text{map}(B_{j_1 \cdots j_n}, 1),$$

where $\mid\!=$ denotes Boolean OR reduction. The computation above logically iterates over every component of B, computes the coordinates $(i_1, \ldots, i_m)$ of each component $B_{j_1 \cdots j_n}$ in the remapped tensor A, and sets the corresponding component in the Boolean result tensor Q to true (1). (All components of Q are assumed to be initialized to false.) The map operator returns the second argument if the first argument is nonzero (or true) and zero otherwise, which ensures only the coordinates of nonzeros in B are aggregated. So if, for instance, C is a $K \times I \times J$ tensor obtained by applying the remapping `(i,j) -> (j-i,i,j)` to a matrix D, then to compute `select [k] -> id() as Q` on C, our technique lowers the query to the computation $\forall_i \forall_j \; Q_{j-i} \mid\!= \text{map}(D_{ij}, 1)$. For each nonzero of D, this computation computes the nonzero's offset from the main diagonal and sets the corresponding component in Q to true. The query result Q thus strictly encodes the set of diagonals in D that contain nonzeros.

In a similar way, our technique lowers `count` queries

```
select [i1,...,im] -> count(im+1,...,il) as Q
```

on A to their canonical form

$$\begin{aligned} &\left(\forall_{i_1} \cdots \forall_{i_l} \; Q_{i_1 \cdots i_m} \mathrel{+}= \text{map}(W_{i_1 \cdots i_l}, 1)\right) \\ &\textbf{where } \left(\forall_{j_1} \cdots \forall_{j_n} \; W_{i_1 \cdots i_l} \mid\!= \text{map}(B_{j_1 \cdots j_n}, 1)\right). \end{aligned}$$

The computation above first iterates over the nonzeros of B to compute the intermediate result W, which encodes whether each subtensor of A identified by coordinates $(i_1, \ldots, i_l)$ is nonzero. The computation then sums over dimensions I_{m+1} through I_l of W to compute the number of aforementioned subtensors that are nonzero and contained in each higher-order subtensor with coordinates $(i_1, \ldots, i_m)$.

Our technique also generates code for `max` queries

```
select [i1,...,im] -> max(im+1) as Q
```

by lowering them to their canonical form

$$\forall_{j_1} \cdots \forall_{j_n} \; Q'_{i_1 \cdots i_m} \; \text{max}\!= \text{map}(B_{j_1 \cdots j_n}, i_{m+1} - s + 1),$$

where s denotes the smallest possible coordinate along dimension I_{m+1}. Q' is assumed to be initialized to the zero tensor, so by mapping each input tensor component to its remapped coordinate i_{m+1} plus the constant $(1-s)$, we ensure that only the coordinates of nonzeros are actually aggregated. Q' can thus be interpreted as the actual result of the original query (i.e., Q) but just shifted by $(1 - s)$; in other words, $Q_{i_1 \cdots i_m} \equiv Q'_{i_1 \cdots i_m} + s - 1$. Similarly, `min` queries

```
select [i1,...,im] -> min(im+1) as Q
```

Table 1. Example transformations that our technique applies to optimize attribute queries. We augment level formats with a property that specifies if a dimension stores explicit zeros, which lets our technique determine if a tensor stores only nonzeros.

Transformation	Definition	Preconditions and Postconditions
reduction-to-assign	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \oplus\!= expr)$ $\implies (\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} = expr)$	For each j_k, there exists an i_l such that $j_k \equiv i_l$. $\oplus$ is any reduction operator. A is initialized to the zero tensor.
inline-temporary	$(\forall_{i_1} \cdots \forall_{i_m} A_{i_1 \cdots i_l} \oplus\!= f(W_{i_1 \cdots i_m}))$ **where** $(\forall_{j_1} \cdots \forall_{j_n} W_{i_1 \cdots i_m} = expr)$ $\implies (\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_l} \oplus\!= f(expr))$	f is any function that takes only W as tensor operand. $\oplus$ is any reduction operator or a simple assignment.
simplify-width-count	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \mathrel{+}= \text{map}(B_{j_1 \cdots j_n}, c))$ $\implies (\forall_{j_1} \cdots \forall_{j_{n-1}} A_{i_1 \cdots i_m} \mathrel{+}= B'_{j_1 \cdots j_{n-1}} \cdot c)$	B stores only nonzeros, and j_n is a reduction variable that indexes into the innermost dimension of B (i.e., J_n). c is any constant. B' is a tensor that encodes the number of nonzeros in each slice of B indexed by coordinates (j_1, …, j_{n-1}); values of B' are not materialized but dynamically computed with calls to level functions that define iteration over dimension J_n of B.
counter-to-histogram	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \text{ max}\!= \text{map}(B_{j_1 \cdots j_n}, \#j_k \cdots j_l + 1))$ $\implies (\forall_{i_1} \cdots \forall_{i_l} A_{i_1 \cdots i_m} \text{ max}\!= W_{i_1 \cdots i_l})$ **where** $(\forall_{j_1} \cdots \forall_{j_n} W_{i_1 \cdots i_m j_k \cdots j_l} \mathrel{+}= \text{map}(B_{j_1 \cdots j_n}, 1))$	None.

are lowered to their canonical form

$$\forall_{j_1} \cdots \forall_{j_n} Q'_{i_1 \cdots i_m} \text{ max}\!= \text{map}(B_{j_1 \cdots j_n}, -i_{m+1} + t + 1),$$

where t denotes the largest possible coordinate along dimension I_{m+1} and Q' is the query result but negated and shifted by $(t + 1)$; in other words, $Q_{i_1 \cdots i_m} \equiv -Q'_{i_1 \cdots i_m} + t + 1$.

After an attribute query is lowered to its canonical form, our technique eagerly applies a set of predefined transformations on the query computation to optimize its performance. Table 1 shows a subset of transformations that our technique uses. In general, these transformations exploit properties of the input tensor and its underlying storage format to reduce the number of dimensions that have to be iterated and to eliminate redundant temporaries.

To see how our technique optimizes attribute queries, consider the example query `select [i] -> count(j) as Q` applied to an $I \times J$ matrix B. As described before, our technique first lowers this query to its canonical form

$$\left(\forall_i \forall_j Q_i \mathrel{+}= \text{map}(W_{ij}, 1)\right) \textbf{ where } \left(\forall_i \forall_j W_{ij} \mid\!= \text{map}(B_{ij}, 1)\right).$$

Our technique then proceeds to iteratively and eagerly apply the transformations shown in Table 1 on the computation above. In particular, each iteration variable bound to a $\forall$ is used to independently index into a dimension of W, so the substatement that defines W satisfies the preconditions of the reduction-to-assign transformation. Our technique thus applies the aforementioned transformation on the substatement that defines W to obtain

$$\left(\forall_i \forall_j Q_i \mathrel{+}= \text{map}(W_{ij}, 1)\right) \textbf{ where } \left(\forall_i \forall_j W_{ij} = \text{map}(B_{ij}, 1)\right).$$

Then, since the temporary W is no longer the result of a reduction operation, our technique eliminates it by applying the inline-temporary transformation to obtain

$$\forall_i \forall_j Q_i \mathrel{+}= \text{map}(\text{map}(B_{ij}, 1), 1),$$

which is then trivially rewritten to $\forall_i \forall_j Q_i \mathrel{+}= \text{map}(B_{ij}, 1)$ by applying constant folding. If B is stored in COO, then we can directly apply the techniques of Kjolstad et al. and Chou et al. to lower this rewritten statement down to imperative code shown on lines 1–6 in Figure 6c. However, if B is stored in CSR (with only nonzeros stored), then our technique additionally applies the simplify-width-count transformation followed by reduction-to-assign again to get the final query

$$\forall_i Q_i = B'_i,$$

where each component of B' is dynamically computed as `pos[i+1] - pos[i]`. The optimized query thus avoids iterating over B's nonzeros, thereby reducing memory traffic.

6 Sparse Tensor Assembly

As explained in Section 2, a sparse tensor can be modeled as a hierarchical structure of coordinates, where each stored component is represented by a path from the root to a leaf. We can thus view any tensor format simply as a composition of level formats that each stores a level of a coordinate hierarchy. This abstraction lets us reason about sparse tensor assembly as coordinate hierarchy construction.

We extend the coordinate hierarchy abstraction with new primitives (level functions) that describe how each level can be efficiently constructed (assembled). These new level functions, unlike analogous ones proposed by Chou et al. [17], describe how coordinates and edges can be efficiently inserted into a coordinate hierarchy assuming certain statistics about the input tensor have been precomputed.

Figures 7 and 11 show how level formats that use disparate data structures can implement the new assembly level functions. All these implementations expose the same static interface, which lets our code generator reason about and emit efficient code to convert tensors between a wide range

```
init_coords(sz_{k-1}, Q_k):
  perm = malloc(N_k - M_k, int);
  K = 0;
  for (i = M_k; i < N_k; i++) {
    if (Q_k[0][i].nz)
      perm[K++] = i;
  }

init_get_pos(sz_{k-1}):
  rperm = malloc(N_k - M_k, int);
  for (i = 0; i < K; i++)
    rperm[perm[i] - M_k] = i;

get_size(sz_{k-1}):
  return sz_{k-1} * K;

get_pos(p_{k-1}, i_1, ..., i_k):
  return p_{k-1} * K + rperm[i_k - M_k];

finalize_get_pos(sz_{k-1}):
  free(rperm);

Q_k := [select [i_k] -> id() as nz]
```

```
unseq_init_edges(sz_{k-1}, Q_k):
  pos = calloc(sz_{k-1} + 1, int);

seq_init_edges(sz_{k-1}, Q_k):
  pos = malloc(sz_{k-1} + 1, int);
  pos[0] = 0;

init_coords(sz_{k-1}, Q_k):
  crd = malloc(pos[sz_{k-1}]], int);

get_size(sz_{k-1}):
  return pos[sz_{k-1}];

unseq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k):
  pos[p_{k-1} + 1] = q_k[0].nir;

seq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k):
  pos[p_{k-1} + 1] = pos[p_{k-1}] + q_k[0].nir;

insert_coord(p_{k-1}, p_k, i_1, ..., i_k):
  crd[p_k] = i_k;

Q_k := [select [i_1, ..., i_{k-1}] ->
        count(i_k) as nir]

unseq_finalize_edges(sz_{k-1}):
  prefix_sum(pos, sz_{k-1} + 1);

yield_pos(p_{k-1}, i_1, ..., i_k):
  return pos[p_{k-1}]++;

finalize_yield_pos(sz_{k-1}):
  for (i = 0; i < sz_{k-1}; i++)
    pos[sz_{k-1} - i] = pos[sz_{k-1} - i - 1];
  pos[0] = 0;
```

```
unseq_init_edges(sz_{k-1}, Q_k):
  pos = calloc(sz_{k-1} + 1, int);

seq_init_edges(sz_{k-1}, Q_k):
  pos = malloc(sz_{k-1} + 1, int);
  pos[0] = 0;

get_pos(p_{k-1}, i_1, ..., i_k):
  return pos[p_{k-1} + 1] + i_k - i_{k-1} - 1;

unseq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k):
  pos[p_{k-1} + 1] = max(i_{k-1} - q_k[0].w + 1, 0);

seq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k):
  pos[p_{k-1} + 1] = pos[p_{k-1}] + max(i_{k-1} - q_k[0].w + 1, 0);

get_size(sz_{k-1}):
  return pos[sz_{k-1}];

unseq_finalize_edges(sz_{k-1}):
  prefix_sum(pos, sz_{k-1} + 1);

Q_k := [select [i_1, ..., i_{k-1}] ->
        min(i_k) as w]
```

Figure 11. Implementations of the assembly abstraction, including level function definitions and the required attribute queries, for three different level formats: *squeezed* (top), *compressed* (middle), and *banded* (bottom). *Squeezed* stores the dimension of offsets in (remapped) DIA tensors. *Compressed* stores the column dimension of CSR tensors as well as the row dimension of COO tensors. *Banded* stores the column dimension of tensors in the skyline format [24], which stores all components between the first nonzero and the diagonal for every row. M_k and N_k denote the lower and upper bounds of the k-th dimension.

of formats. And by using the same level formats to express different tensor formats that share common data structures, our technique can reuse the same level function implementations to generate conversion routines for many different target formats. For example, the column dimensions of CSR tensors and the row dimensions of COO tensors can both be stored using the same (i.e., compressed) level format. The code generator can thus use the same implementations of the new assembly level functions (for the compressed level format) to generate parts of code that convert tensors to either CSR or COO. This limits the one-time effort needed to implement our extended coordinate hierarchy abstraction.

6.1 Assembly Abstraction

Our extended coordinate hierarchy abstraction assumes that coordinate hierarchies can be constructed level by level from top to bottom. The assembly of each level is decomposed into two logical phases: *edge insertion* and *coordinate insertion*.

The edge insertion phase, which is optional, logically bulk inserts edges into a coordinate hierarchy to connect coordinates in adjacent levels. Edge insertion models the assembly of data structures that map nonzeros to their containing subtensors. Depending on whether each position (node) in the preceding parent level can be iterated in sequence, edge insertion can be done in a *sequenced* or *unsequenced* fashion.

Unsequenced edge insertion is defined in terms of three level functions that any level format may implement:

- `unseq_init_edges(sz`$_{k-1}$`, Q`$_k$`) -> void`
- `unseq_insert_edges(p`$_{k-1}$`, i`$_1$`, ..., i`$_{k-1}$`, q`$_k$`) -> void`
- `unseq_finalize_edges(sz`$_{k-1}$`) -> void`

Q_k denotes the (complete) results of attribute queries that a level format specifies must be precomputed, while q_k denotes only the elements of Q_k indexed by coordinates $(i_1, \ldots, i_{k-1})$. sz_{k-1} is the size of the parent level and can be computed as a function of its own parent's size by calling the level function

`get_size(sz`$_{k-1}$`) -> sz`$_k$`,`

which all level formats must also implement. `unseq_init_edges` initializes data structures that the level format uses to logically store edges. Then, for each position p_{k-1} (which represents a subtensor with coordinates $(i_1, \ldots, i_{k-1})$) in the parent level, `unseq_insert_edges` allocates some number of child coordinates to be connected to p_{k-1}. The number of child coordinates allocated can be computed as any arbitrary function of q_k. Finally, `unseq_finalize_edges` inserts edges into the coordinate hierarchy so that each coordinate in the parent level is connected to as many children as it was previously allocated. Figure 12 (left) shows how these level functions can logically be invoked to bulk insert edges.

```
sz_{k-1} = get_size_{k-1}(get_size_{k-2}(···(1)···));
unseq_init_edges(sz_{k-1}, Q_k);
for (position p_{k-1} in parent level | ∃ coords
     i_1, ..., i_{k-1} connecting p_{k-1} to root) {
  q_k[:] = Q_k[:][i_1, ..., i_{k-1}];
  unseq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k);
}
unseq_finalize_edges(sz_{k-1});
```

```
init_coords(sz_{k-1}, Q_k);
init_{get|yield}_pos(sz_{k-1});
for (nonzero with coords i_1, ..., i_k) {
  for (j = 1; j <= k; j++) // can be unrolled
    p_j = {get|yield}_pos_j(p_{j-1}, i_1, ..., i_j);
  insert_coord(p_{k-1}, p_k, i_1, ..., i_k);
}
finalize_{get|yield}_pos(sz_{k-1});
```

Figure 12. Unsequenced edge insertion (left) and coordinate insertion (right), expressed in terms of calls to level functions.

Sequenced edge insertion, by contrast, is defined in terms of just two level functions:

- `seq_init_edges(sz`$_{k-1}$`, Q`$_k$`) -> void`
- `seq_insert_edges(p`$_{k-1}$`, i`$_1$`, ..., i`$_{k-1}$`, q`$_k$`) -> void`

These level functions are analogous to `unseq_init_edges` and `unseq_insert_edges` and can be invoked in similar ways. Sequenced edge insertion, however, assumes that all positions in the parent level are iterated in order. Thus, `seq_insert_edges` is responsible for both allocating the appropriate number of child coordinates to each parent and actually inserting the edges, and a separate `finalize` function is not necessary.

The coordinate insertion phase logically iterates over the input tensor's nonzeros and inserts their coordinates into a coordinate hierarchy. This phase models the assembly of data structures that store the coordinates and values of the nonzeros, and it is defined in terms of five level functions:

- `init_coords(sz`$_{k-1}$`, Q`$_k$`) -> void`
- `init_{get|yield}_pos(sz`$_{k-1}$`) -> void`
- `{get|yield}_pos(p`$_{k-1}$`, i`$_1$`, ..., i`$_k$`) -> p`$_k$
- `insert_coord(p`$_{k-1}$`, p`$_k$`, i`$_1$`, ..., i`$_k$`) -> void`
- `finalize_{get|yield}_pos(sz`$_{k-1}$`) -> void`

`init_coords` allocates and initializes data structures for storing coordinates in a coordinate hierarchy level. If a level format implicitly encodes coordinates (e.g., as a fixed range) using some fixed set of parameters, then `init_coords` also compute those parameters as functions of the attribute query results Q_k. On the other hand, if a level format explicitly stores coordinates in memory, then the coordinates of nonzeros are inserted by invoking `insert_coord` for each nonzero. The position p_k at which each nonzero should be inserted is computed by invoking either `get_pos` or `yield_pos`. The former guarantees that nonzeros with the same coordinates are inserted at the same position. The latter allows duplicate coordinates to be inserted at different positions. Both functions, however, may rely on auxiliary data structures to track where to insert coordinates; `init_{get|yield}_pos` and `finalize_{get|yield}_pos` initializes and cleans up those data structures. Figure 12 (right) shows how all these level functions can be invoked to perform coordinate insertion.

6.2 Code Generation

To generate code that converts sparse tensors between two formats, our code generator emits loops that iterate over a tensor in the source format and apply the target format's coordinate remapping to each nonzero. This is done by applying the technique described in Section 4.2. Then, within each loop nest that iterates over the (remapped) input tensor, the code generator emits code that invokes the level functions described in Section 6.1 to store each nonzero into the result. The emitted code is finally specialized to the target format by inlining its implementations of the aforementioned level functions (e.g., as shown in Figure 11). This approach enables the code generator to support disparate target (and source) formats. At the same time, it limits the complexity of the code generator, since the code generator does not need to hard-code for specific data structures but can simply reason about how to invoke a fixed set of level functions.

In order to minimize memory traffic at runtime, our technique emits code that, wherever possible, fuses the assembly of adjacent levels in the result coordinate hierarchy. Adjacent levels can be assembled together as long as only the parent level requires a separate edge insertion phase (or if none do). As an example, none of the level formats that compose DIA requires edge insertion. Thus, our technique will emit code to convert any matrix to DIA by iterating over the matrix just once and assembling all output dimensions (levels) together.

For each set of levels that can be assembled together, our technique then simply has to emit code like what is shown in Figure 12 to perform edge insertion (if required) followed by coordinate insertion. If a level format implements both variants of edge insertion, then our technique selects one based on whether the parent level can be iterated in order. The code generator infers this from properties exposed through the coordinate hierarchy abstraction that specify if the parent level stores coordinates in order. If a level format implements `yield_pos` but does not permit storing duplicates of the same coordinate, our technique also emits logic to perform deduplication on the fly by keeping track of inserted coordinates.

To see how our technique works, suppose we are generating code to convert COO tensors to CSR. To obtain code that assembles the column dimension of the result, the code generator first emits sequenced edge insertion code that has the same structure as Figure 12 (left), except with all level functions replaced by their sequenced counterparts. The emitted code is then specialized to CSR by replacing the level function calls with the compressed level format's implementations of those functions (Figure 11, middle). The result

Table 2. Statistics about matrices used in our experiments. Non-symmetric matrices are highlighted in gray.

Matrix	Dimensions	NNZ	Nonzero Diagonals	Max. NNZ/row
pdb1HYS	36.4K × 36.4K	4.34M	26K	204
jnlbrng1	40.0K × 40.0K	199K	5	5
obstclae	40.0K × 40.0K	199K	5	5
chem_master1	40.4K × 40.4K	201K	5	5
rma10	46.8K × 46.8K	2.37M	17K	145
dixmaanl	60.0K × 60.0K	300K	7	5
cant	62.5K × 62.5K	4.01M	99	78
shyy161	76.5K × 76.5K	330K	7	6
consph	83.3K × 83.3K	6.01M	13K	81
denormal	89.4K × 89.4K	1.16M	13	13
Baumann	112K × 112K	748K	7	7
cop20k_A	121K × 121K	2.62M	221K	81
shipsec1	141K × 141K	3.57M	10K	102
majorbasis	160K × 160K	1.75M	22	11
scircuit	171K × 171K	959K	159K	353
mac_econ_fwd500	207K × 207K	1.27M	511	44
pwtk	218K × 218K	11.5M	20K	180
Lin	256K × 256K	1.77M	7	7
ecology1	1.00M × 1.00M	5.00M	5	5
webbase-1M	1.00M × 1.00M	3.11M	564K	4700
atmosmodd	1.27M × 1.27M	8.81M	7	7

is lines 7–11 in Figure 6c, which iterate over all rows of the output in order and reserve enough memory to store each row's nonzeros. In a similar way, the code generator emits code in Figure 12 (right) to perform coordinate insertion and then specializes it to CSR, yielding lines 12–25 in Figure 6c.

7 Evaluation

We evaluate our technique and find that it generates efficient sparse tensor conversion routines for many combinations of disparate source and target formats. The generated conversion routines have performance similar to or better than equivalent hand-implemented versions. We also find that, for combinations of source and target formats that are not directly supported by a library, our technique can further optimize conversions between those formats by emitting code that avoids materializing temporaries.

7.1 Experimental Setup

We implemented a prototype of our technique as extensions to the open-source taco tensor algebra compiler [30]. To evaluate it, we compare code that our technique generates against hand-implemented versions in SPARSKIT [48], a widely used [25, 42] Fortran sparse linear algebra library, and Intel MKL [24], a C and Fortran math processing library that is optimized for Intel processors. We also evaluate our technique against taco without our extensions.[1]

All experiments are conducted on a 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. We compile code that our technique generates using GCC 7.5.0 and build SPARSKIT from source using GFortran 7.5.0. We run each experiment 50 times under cold cache conditions and report median serial execution times.

We run our experiments with real-world matrices of varying sizes and structures as inputs. These matrices, listed in Table 2, come from applications in disparate domains and are obtained from the SuiteSparse Matrix Collection [18].

7.2 Performance Evaluation

We measure the performance of sparse tensor conversion routines that our technique generates for seven distinct combinations of source and target formats:

- COO to CSR (`coo_csr`)
- COO to DIA (`coo_dia`)
- CSR to CSC (`csr_csc`)
- CSR to DIA (`csr_dia`)
- CSR to ELL (`csr_ell`)
- CSC to DIA (`csc_dia`)
- CSC to ELL (`csc_ell`)

where inputs and outputs in COO, CSR, or CSC are not assumed to be sorted (though nonzeros are still necessarily grouped by row/column in CSR/CSC). For each combination of formats, we also measure the performance of conversion between those formats using SPARSKIT and Intel MKL. Both libraries implement routines that directly convert matrices from COO to CSR, CSR to CSC, and CSR to DIA. Additionally, SPARSKIT supports directly converting matrices from CSR to ELL. However, neither SPARSKIT nor Intel MKL implements routines that directly convert matrices between the remaining combinations of formats. Thus, to perform those conversions using either library, we first convert the input from its source format to a temporary in CSR and then convert the temporary from CSR to the intended target format. (If the input matrix is symmetric though, then conversions from CSC to DIA/ELL are cast as conversions from CSR to DIA/ELL, since CSR and CSC are equivalent in that case.)

Table 3 show the results of our experiments. As these results demonstrate, our technique outperforms or is comparable to SPARSKIT and Intel MKL on average for all combinations of source and target formats that we evaluate. On the whole, code that our technique emits to convert matrices from COO to CSR (`coo_csr`) and from CSR to CSC (`csr_csc`) exhibit similar performance as hand-implemented routines in SPARSKIT and somewhat better performance than Intel MKL. This is unsurprising since our technique generates code that implement the same algorithms (variations of Gustavson's HALFPERM algorithm [22]) as SPARSKIT. Our technique also emits code to perform CSR to DIA conversion that is 2.01× faster than SPARSKIT and 1.80× faster than Intel MKL on average. SPARSKIT's implementation of `csr_dia` supports extracting a bounded number of nonzero diagonals from the input matrix and storing them in the output. However, it implements this capability using an inefficient algorithm to identify the densest diagonals, thus leading to the

[1] https://github.com/tensor-compiler/taco/tree/c0e93b65

Table 3. Normalized execution times of conversion routines that are implemented or generated in SPARSKIT (skit), Intel MKL (mkl) and taco without our extensions (taco w/o ext.), relative to code that our technique generates (taco w/ ext.). The actual execution times (in milliseconds) of code that our technique generates are shown in parentheses. For CSR to CSC conversion, we only show results for nonsymmetric matrices (rows highlighted in gray) since CSR and CSC are equivalent for symmetric matrices. For symmetric matrices, we cast CSC to DIA/ELL conversion as CSR to DIA/ELL conversion and report the same results. For conversions to DIA/ELL, we also omit results for matrices that would have to be stored with more than 75% of values being zeros, since having to compute with so many zeros would likely eliminate any performance benefit of DIA/ELL. Columns highlighted in gray denote kernels that perform the conversion by first converting to CSR temporaries.

Matrix	coo_csr				coo_dia			csr_csc			csr_dia			csr_ell		csc_dia			csc_ell	
	taco w/ ext.	taco w/o ext.	skit	mkl	taco w/ ext.	skit	mkl	taco w/ ext.	skit	mkl	taco w/ ext.	skit	mkl	taco w/ ext.	skit	taco w/ ext.	skit	mkl	taco w/ ext.	skit
pdb1HYS	1 (57.5)	15.88	1.02	1.11										1 (79.1)	1.68				1 (79.1)	1.68
jnlbrng1	1 (0.96)	31.15	0.97	1.56	1 (0.86)	3.07	3.38				1 (0.91)	1.85	1.56	1 (0.92)	0.95	1 (0.91)	1.85	1.56	1 (0.92)	0.95
obstclae	1 (0.93)	31.95	1.00	1.60	1 (0.86)	3.02	3.36				1 (0.91)	1.84	1.54	1 (0.82)	1.04	1 (0.91)	1.84	1.54	1 (0.82)	1.04
chem_master1	1 (1.06)	29.05	1.04	1.44	1 (0.88)	4.60	4.94	1 (1.10)	0.98	2.14	1 (0.93)	1.83	1.54	1 (0.91)	0.91	1 (0.96)	4.24	3.85	1 (1.24)	3.11
rma10	1 (34.0)	12.77	1.01	0.96				1 (29.1)	1.17	1.16				1 (49.2)	1.84				1 (62.8)	2.09
dixmaanl	1 (1.61)	30.64	1.02	1.42	1 (1.50)	5.03	3.11				1 (1.54)	1.88	1.57	1 (1.35)	0.97	1 (1.54)	1.88	1.57	1 (1.35)	0.97
cant	1 (27.4)	25.89	1.00	1.35	1 (45.3)	4.16	4.39				1 (44.5)	3.61	3.63	1 (59.8)	1.78	1 (44.5)	3.61	3.63	1 (59.8)	1.78
shyy161	1 (1.69)	26.92	1.00	1.50	1 (1.76)	4.98	3.05	1 (1.64)	1.07	2.36	1 (1.86)	1.85	1.54	1 (1.77)	0.94	1 (1.89)	4.68	3.44	1 (2.36)	3.00
consph	1 (64.8)	18.58	1.01	1.21										1 (88.9)	1.45				1 (88.9)	1.45
denormal	1 (5.61)	33.13	1.01	1.51	1 (5.14)	5.10	5.78				1 (4.83)	2.21	2.26	1 (5.17)	1.02	1 (4.83)	2.21	2.26	1 (5.17)	1.02
Baumann	1 (4.70)	25.21	0.99	1.49	1 (3.48)	5.23	5.48	1 (4.71)	1.03	1.89	1 (3.56)	1.95	1.70	1 (3.33)	1.01	1 (3.60)	5.07	4.10	1 (4.74)	3.16
cop20k_A	1 (63.6)	8.46	0.89	0.96										1 (34.8)	3.60				1 (34.8)	3.60
shipsec1	1 (81.6)	18.29	1.02	1.28										1 (112)	1.93				1 (112)	1.93
majorbasis	1 (12.3)	24.05	1.00	1.33	1 (10.9)	3.34	3.70	1 (12.1)	0.99	1.78	1 (9.91)	2.43	2.42	1 (8.17)	1.03	1 (10.4)	3.47	4.37	1 (20.1)	1.88
scircuit	1 (15.8)	11.54	1.00	1.10				1 (16.4)	0.95	1.09										
mac_econ_fwd500	1 (11.1)	20.52	0.99	1.29				1 (11.6)	1.00	1.38										
pwtk	1 (121)	19.77	1.00	1.29										1 (123)	4.09				1 (123)	4.09
Lin	1 (9.88)	30.12	0.98	1.36	1 (8.40)	4.87	5.12				1 (8.14)	1.92	1.70	1 (10.1)	0.98	1 (8.14)	1.92	1.70	1 (10.1)	0.98
ecology1	1 (42.3)	19.82	0.99	1.41	1 (36.8)	2.74	3.00				1 (35.8)	1.64	1.44	1 (37.5)	1.08	1 (35.8)	1.64	1.44	1 (37.5)	1.08
webbase-1M	1 (57.9)	10.27	1.01	0.99				1 (59.3)	1.00	1.14										
atmosmodd	1 (113)	15.15	0.96	1.21	1 (64.9)	3.26	3.49	1 (113)	0.97	1.04	1 (62.2)	1.72	1.58	1 (74.1)	1.17	1 (62.9)	3.40	3.39	1 (88.7)	2.20
Geomean	**1**	**20.39**	**1.00**	**1.29**	**1**	**4.01**	**3.96**	**1**	**1.02**	**1.48**	**1**	**2.01**	**1.80**	**1**	**1.36**	**1**	**2.75**	**2.51**	**1**	**1.78**

slowdown. In addition, `csr_ell` code that our technique emits is 1.36× faster than SPARSKIT on average. This is because our technique emits code that invokes `calloc` to both allocate and zero-initialize the output arrays. SPARSKIT, by contrast, takes user-allocated output arrays as arguments and separately initializes those arrays. Furthermore, for conversions to DIA and ELL, code that our technique emits achieve even greater speedups—between 1.78 and 4.01×—over SPARSKIT and Intel MKL when converting from COO or CSC. This is because when the input matrix is nonsymmetric or stored in COO, both libraries must incur additional memory accesses to construct and then iterate over temporary CSR matrices.

Finally, we measure and compare against the performance of taco without our extensions for COO to CSR conversion. By expressing COO to CSR conversion in index notation as tensor assignment (i.e., $A_{ij} = B_{ij}$, where A and B are CSR and COO matrices respectively), the techniques of Kjolstad et al. and Chou et al. [17, 29, 30] can also generate code that performs the conversion. As Table 3 also shows though, our technique emits code to perform COO to CSR conversion that is 20.4× faster on average. The techniques of Kjolstad et al. and Chou et al. cannot reason about generating code that inserts nonzeros into CSR data structures out of order. Thus, it must sort the input before performing the actual conversion, incurring significant additional overhead. Furthermore, while sparse matrix formats like DIA or ELL can be cast as 3rd-order tensor formats, index notation (as described in [17, 29, 30]) cannot express assignment of a matrix to a 3rd-order tensor in a way that does not duplicate nonzeros along the extra dimension. So without the extensions described in this paper, taco, which takes index notation as input, cannot emit code to perform end-to-end conversion for formats that store nonzeros in non-lexicographic coordinate order.

8 Related Works

There is a long line of works [5, 8, 10, 13, 23, 34–36, 47, 50, 60] on developing new sparse tensor formats to accelerate SpMV, sparse matrix-dense matrix multiplication (SpDM/SpMM), matricized tensor times Khatri-Rao products (MTTKRP), and other tensor computations. These formats organize nonzeros in disparate ways to reduce memory footprint, improve cache utilization, expose parallelization opportunities, and better

exploit hardware capabilities such as SIMD vector units for performance. All the aforementioned works rely on hand-implemented routines for converting tensors from a standard representation (e.g., COO) to their proposed formats. These routines are often more complex than code to perform the computations that the proposed formats are designed to accelerate. In addition, many techniques [15, 21, 22, 57–59] have been proposed for accelerating transpositions of CSR matrices, which is equivalent to CSR to CSC conversion.

Existing sparse tensor and linear algebra compilers cannot generate efficient code to convert tensors between arbitrary, disparate formats. The taco compiler [17, 29, 30] can emit code to convert tensors between formats that store nonzeros in lexicographic coordinate order, but cannot generate complete conversion routines for structured formats like DIA and ELL. Without the extensions described in this paper, taco also cannot emit code that computes and uses statistics about the input tensor to coordinate efficient assembly of the output tensor. LL [3, 4] is a functional language that lets users define sparse matrix formats as nestings of lists and pairs that encode nonzeros in a dense matrix. From these specifications, the LL compiler can emit code that convert dense matrices to different sparse matrix formats, but not efficient code that can directly convert between sparse matrix formats. In the context of inspector-executor approaches for sparse linear algebra, Nandy et al. [39] build on CHiLL [56] and SPF [52] to show how inspectors that convert input matrices between different sparse matrix formats can be generated. Their approach, however, requires specifications to be provided for every combination of potential source and target formats, since each specification is hard-coded to data structures used by the source and target formats. SIPR [45] and techniques that Bik and Wijshoff proposed [11, 12] can also generate sparse linear algebra code that, as sub-operations, convert matrices between different formats. However, conversions in SIPR-generated code are performed by invoking hand-implemented operations that are hard-coded to specific source and target formats. Meanwhile, the techniques proposed by Bik and Wijshoff only support a fixed set of standard sparse matrix formats. Bernoulli [31, 32, 51] uses a black-box protocol that provides an abstract interface for describing how sparse matrices stored in different data structures can be efficiently accessed. However, the interface does not support assembly, so Bernoulli cannot generate code that construct sparse matrix results.

There also exists a separate line of works [16, 26, 33, 40] on generating efficient code for query languages like SQL, which our attribute query language resembles. (Attribute queries are analogous to GROUP BY queries on a table that stores the coordinates of nonzeros.) In particular, HorseIR [16] lowers SQL queries to an array-based intermediate representation that is then optimized and compiled to efficient code. EmptyHeaded [1] is a graph processing framework that generates efficient code to compute graph queries expressed in a Datalog-like language. Furthermore, our approach to optimizing attribute queries is reminiscent of query rewriting systems in certain relational database systems like Starburst [43, 44]. All these techniques are designed for queries that may perform complex joins and aggregate data of arbitrary types. By contrast, attribute queries are limited to aggregating tensor coordinates, which are integers. This lets our technique lower and optimize attribute queries in ways that would be invalid for aggregations over other data types.

9 Conclusion and Future Work

We have described a technique for generating sparse tensor conversion routines that efficiently convert tensors between a wide range of formats. Our technique is extensible, so users can easily add support for new source and target formats by simply specifying how to construct and iterate over tensors in those new formats. By making it easy to work with the same data in multiple formats, each suited to a different stage of an application, our technique can greatly reduce the effort needed to optimize sparse tensor algebra applications.

That said, our technique can be further generalized in various ways to support conversions between an even wider range of formats. A limitation of our technique is that it only supports tensor formats that can be expressed in the coordinate hierarchy abstraction we proposed in [17]. This, for instance, precludes support for conversions to hybrid formats like HYB [9] that decompose a tensor into multiple subtensors and store each using a different format, which the abstraction does not support. Coordinate remapping notation as we proposed also does not support grouping nonzeros based on statistics of the input tensor. Such a capability could again, for instance, be useful for supporting conversions to hybrid tensor formats, which require determining how to divide up nonzeros amongst different subtensors. Furthermore, our technique as described does not support generating fully parallelized code for CPUs or GPUs. We believe addressing such limitations would constitute valuable future work.

Acknowledgments

We thank Peter Ahrens, Rawn Henry, Ziheng Wang, Michelle Strout, and other anonymous reviewers for their helpful reviews and suggestions. This work was supported by the Application Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; the Toyota Research Institute; the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0008923 and DE-SC0018121; the National Science Foundation under Grant No. CCF-1533753; and DARPA under Awards HR0011-18-3-0007 and HR0011-20-9-0017. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246

[2] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. 2014. Tensor Decompositions for Learning Latent Variable Models. *J. Mach. Learn. Res.* 15, Article 1 (Jan. 2014), 60 pages.

[3] Gilad Arnold. 2011. *Data-Parallel Language for Correct and Efficient Sparse Matrix Codes.* Ph.D. Dissertation. University of California, Berkeley.

[4] Gilad Arnold, Johannes Hölzl, Ali Sinan Köksal, Rastislav Bodík, and Mooly Sagiv. 2010. Specifying and Verifying Sparse Matrix Codes. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1863543.1863581

[5] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. 2014. An Efficient Two-dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/2597652.2597678

[6] Brett W. Bader, Michael W. Berry, and Murray Browne. 2008. *Discussion Tracking in Enron Email Using PARAFAC.* Springer London, 147–163.

[7] Brett W Bader and Tamara G Kolda. 2007. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2007), 205–231.

[8] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. 2012. Efficient and scalable computations with sparse tensors. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–6. https://doi.org/10.1109/HPEC.2012.6408676

[9] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation.

[10] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[11] Aart JC Bik and Harry AG Wijshoff. 1993. Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing*. ACM, 416–424.

[12] Aart JC Bik and Harry AG Wijshoff. 1994. On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing*. Springer, 57–75.

[13] Aydin 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 *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 233–244.

[14] Aydin Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS)*. 1–11.

[15] Frank Cameron. 1993. Two space-saving algorithms for computing the permuted transpose of a sparse matrix. *Advances in Engineering Software* 17, 1 (Jan. 1993), 49–60. https://doi.org/10.1016/0965-9978(93)90041-Q

[16] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. 2018. HorseIR: Bringing Array Programming Languages Together with Database Query Processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages* (Boston, MA, USA) *(DLS 2018)*. ACM, New York, NY, USA, 37–49. https://doi.org/10.1145/3276945.3276951

[17] 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.

[18] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011).

[19] Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. 2005. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Proceedings of the 5th International Conference on Computational Science - Volume Part I* (Atlanta, GA) *(ICCS'05)*. Springer-Verlag, Berlin, Heidelberg, 99–106. https://doi.org/10.1007/11428831_13

[20] A. Elafrou, G. Goumas, and N. Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. In *2017 46th International Conference on Parallel Processing (ICPP)*. 292–301.

[21] Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez, and Oscar Plata. 2013. Parallelizing the Sparse Matrix Transposition: Reducing the Programmer Effort Using Transactional Memory. *Procedia Computer Science* 18 (2013), 501 – 510. https://doi.org/10.1016/j.procs.2013.05.214 2013 International Conference on Computational Science.

[22] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269. https://doi.org/10.1145/355791.355796

[23] Eun-jin Im and Katherine Yelick. 1998. Model-Based Memory Hierarchy Optimizations for Sparse Matrices. In *In Workshop on Profile and Feedback-Directed Compilation.*

[24] Intel. 2020. Intel Math Kernel Library Developer Reference. https://software.intel.com/sites/default/files/mkl-2020-developer-reference-c.pdf.pdf

[25] Yuanlin Jiang. 2007. *Techniques for Modeling Complex Reservoirs and Advanced Wells.* Ph.D. Dissertation. Stanford University.

[26] Jun Rao, H. Pirahesh, C. Mohan, and G. Lohman. 2006. Compiled Query Execution Engine using JVM. In *22nd International Conference on Data Engineering (ICDE'06)*. 23–23. https://doi.org/10.1109/ICDE.2006.40

[27] David R. Kincaid, Thomas C. Oppe, and David M. Young. 1989. *ITPACKV 2D User's Guide.*

[28] Fredrik Kjolstad. 2020. *Sparse Tensor Algebra Compilation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[29] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. (2019), 180–192. http://dl.acm.org/citation.cfm?id=3314872.3314894

[30] 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

[31] Vladimir Kotlyar. 1999. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs.* Ph.D. Dissertation. Cornell University.

[32] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*. Springer, 318–327.

[33] K. Krikellas, S. D. Viglas, and M. Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 613–624. https://doi.org/10.1109/ICDE.2010.5447892

[34] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 19, 15 pages. https://doi.org/10.1109/SC.2018.00022

[35] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. 2017. A Unified Optimization Approach for Sparse Tensor Operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 47–57. https://doi.org/10.1109/CLUSTER.2017.75

[36] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. ACM, New York, NY, USA,

339–350. https://doi.org/10.1145/2751205.2751209

[37] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

[38] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing*. Technical report.

[39] Payal Nandy, Mary Hall, Eddie C. Davis, Catherine Mills Olschanowsky, Mahdi Soltan Mohammadi, Wei He, and Michelle Mills Strout. 2018. Abstractions for Specifying Sparse Matrix Data Transformations. In *Proceedings of Eighth InternationalWorkshop on Polyhedral Compilation Techniques* (Manchester, United Kingdom) *(IMPACT 2018)*.

[40] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[41] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. arXiv:cs.CV/1608.01409

[42] Andrés Peratta and Viktor Popov. 2006. A new scheme for numerical modelling of flow and transport processes in 3D fractured porous media. *Advances in Water Resources* 29, 1 (2006), 42 – 61. https://doi.org/10.1016/j.advwatres.2005.05.004

[43] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/130283.130294

[44] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. 1997. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE '97)*. IEEE Computer Society, Washington, DC, USA, 391–400. http://dl.acm.org/citation.cfm?id=645482.653436

[45] William Pugh and Tatiana Shpeisman. 1999. SIPR: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing*. Springer, 213–229.

[46] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. 2017. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/3037697.3037745

[47] Youcef Saad. 1989. Krylov Subspace Methods on Supercomputers. *SIAM J. Sci. Stat. Comput.* 10, 6 (Nov. 1989), 1200–1232. https://doi.org/10.1137/0910073

[48] Youcef Saad. 1994. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2.

[49] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[50] 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*. ACM, 5.

[51] Paul Stodghill. 1997. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. Ph.D. Dissertation. Cornell University.

[52] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. 2016. An approach for code generation in the Sparse Polyhedral Framework. *Parallel Comput.* 53 (2016), 32 – 57. https://doi.org/10.1016/j.parco.2016.02.004

[53] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

[54] The SciPy community. 2018. scipy.sparse.dok_matrix – SciPy v1.1.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.dok_matrix.html.

[55] William F Tinney and John W Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

[56] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI 2015)*. 521–532.

[57] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS âĂŹ16)*. Association for Computing Machinery, New York, NY, USA, Article 33, 13 pages. https://doi.org/10.1145/2925426.2926291

[58] Tien-Hsiung Weng, Delgerdalai Batjargal, Hoa Pham, Meng-Yen Hsieh, and Kuan-Ching Li. 2013. Parallel Matrix Transposition and Vector Multiplication Using OpenMP. In *Intelligent Technologies and Engineering Systems (Lecture Notes in Electrical Engineering)*, Jengnan Juang and Yi-Cheng Huang (Eds.). Springer, New York, NY, 243–249. https://doi.org/10.1007/978-1-4614-6747-2_30

[59] Tien-Hsiung Weng, Hoa Pham, Hai Jiang, and Kuan-Ching Li. 2013. Designing Parallel Sparse Matrix Transposition Algorithm Using CSR for GPUs. In *Intelligent Technologies and Engineering Systems (Lecture Notes in Electrical Engineering)*, Jengnan Juang and Yi-Cheng Huang (Eds.). Springer, New York, NY, 251–257. https://doi.org/10.1007/978-1-4614-6747-2_31

[60] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. CVR: Efficient Vectorization of SpMV on x86 Processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 149–162. https://doi.org/10.1145/3168818

[61] Albert-Jan N. Yzelman and Rob H. Bisseling. 2012. A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve. In *Progress in Industrial Mathematics at ECMI 2010*, Michael Günther, Andreas Bartel, Markus Brunk, Sebastian Schöps, and Michael Striebel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 627–633.