Unified Sparse Formats for Tensor Algebra Compilers

by

Stephen Chou

BASc, Computer Engineering, University of Waterloo (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author . . . . . .
Department of Electrical Engineering and Computer Science
January 31, 2018

Certified by . . . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Unified Sparse Formats for Tensor Algebra Compilers

by

Stephen Chou

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Tensor algebra is a powerful tool for computing on multidimensional data and has applications in many fields. Practical applications often deal with tensors that are sparse, and there exists a wide variety of formats for storing such tensors, each suited to specific types of applications and data. Examples of sparse tensor storage formats include COO, CSR, CSC, DCSR, BCSR, CSF, CSB, ELL, DIA, and hash maps.

In this thesis, we propose a levelized hierarchical abstraction that represents these seemingly disparate formats and countless others, and that hides the details of each format behind a common interface. We show that this tensor representation facilitates automatic generation of efficient compute kernels for tensor algebra expressions with any combination of formats. This is accomplished with a code generation algorithm that generates code level by level, guided by the capabilities and properties of the levels.

The performance of tensor algebra kernels generated using our technique is competitive with that of equivalent hand-implemented kernels in existing sparse linear and tensor algebra libraries. Furthermore, our technique can generate many more kernels for many more formats than exist in libraries or are supported by existing compiler techniques.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis includes work published in [34] that was done with Fredrik Kjolstad, Shoaib Kamil, David Lugato, and Saman Amarasinghe. Special thanks goes to the coauthors of the main work presented in this thesis: Prof. Saman Amarasinghe, who is also my research advisor, and Fredrik Kjolstad. They contributed a lot of insightful feedback and ideas that greatly improved the technique described in this thesis, as well as provided significant help with the presentation of the thesis. Their guidance and advice on graduate studies and research in general were also invaluable. I would also like to thank Shoaib Kamil and David Lugato for reviewing various drafts of the thesis and giving valuable feedback.

Additionally, I would like to thank Charith Mendis, Yunming Zhang, and Vladimir Kiriansky for guidance they provided at various points with performance benchmarking and debugging. I would also like to thank them, as well as Riyadh Baghdadi and Jessica Ray, for all the interesting discussions on topics both relating and not relating to research.

Last but not least, I would like to thank my parents, Jennifer Wu and Perry Chou, for the unconditional love and support they have always given me.

Contents

List of Figures

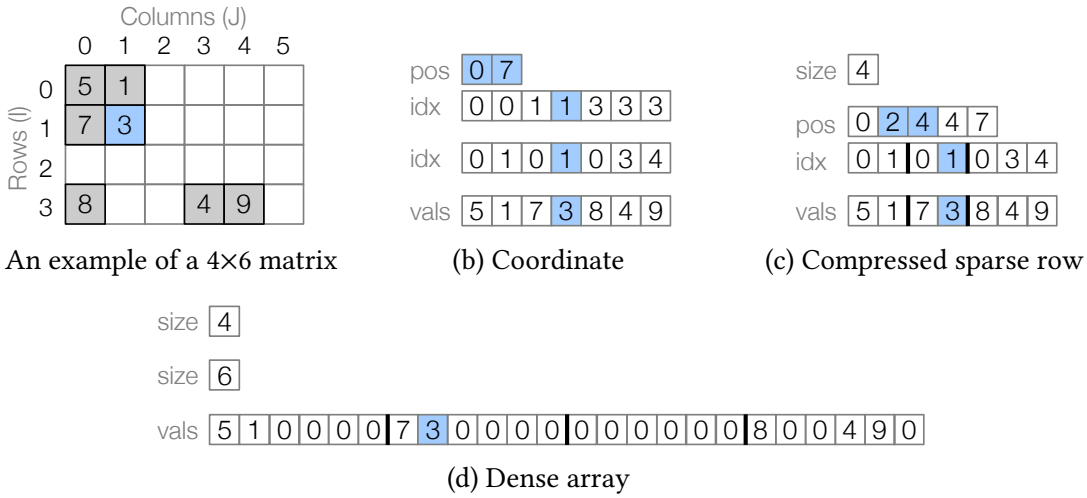
List of Tables

Chapter 1

Introduction

Tensor algebra is a powerful tool for computing on multidimensional data and has practical applications in fields ranging from data analytics and machine learning to the physical sciences and engineering [1, 7, 2, 37, 23, 25, 32]. Tensors generalize matrices to any number of dimensions and model multilinear relationships. Real-world applications often work with tensors that are both very large and extremely sparse, meaning most components are zeros. A tensor encoding product reviews on Amazon [45], for instance, contains 1.5×10^{19} components, but only one in 10^{10} is non-zero. To obtain reasonable computational performance on tensors such as this, it is critical to take advantage of sparsity.

Many formats have been proposed for storing sparse matrices and tensors [68, 33, 55, 16, 15, 58, 10]. The appropriate choice of format depends on how the tensor will be used in an application, on the structure of the data, and on the hardware. In some data analytics applications, one needs to compute using a tensor only once, in which case a format like the coordinate (COO) format that permits efficient assembly and modification can offer the best end-to-end performance. By contrast, in physical simulations and applications that involve tensor factorization, sparse tensors are often reused in multiple iterations of a computation and more space-efficient formats, such as compressed sparse rows (CSR) or compressed sparse fibers (CSF) [58], can significantly reduce storage cost while accelerating memory bandwidth-bound computations. If the tensors also exhibit regular structure, such as in stencil computations or FEM simulations, then a specialized format like the diagonal



(a) An example of a 4×6 matrix

(b) Coordinate

(c) Compressed sparse row

(d) Dense array

Figure 1-1: Examples of the same order-2 tensor (matrix) stored in different formats.

(DIA) format or block CSR (BCSR) can take advantage of hardware features such as SIMD extensions to further boost computational performance.

The existence of all these formats poses a significant challenge when computing with sparse tensors. As Figure 1-1 illustrates, the exact same tensor can be encoded in drastically different ways depending on which format it is stored in. Some tensor formats, like the COO format (Figure 1-1b), explicitly store the coordinates of every non-zero component. Others, like the dense array format (Figure 1-1d), use just a few parameters (e.g., the `size` parameters) to succinctly encode the complete set of coordinates that are assumed to have non-zero values. Additionally, many formats implicitly impose restrictions on how to efficiently iterate non-zeros; the CSR format, for instance, permits efficiently accessing all non-zero columns in any given row (by indexing the `pos` array with the row coordinate) but does not allow efficiently accessing all non-zero rows in a given column. As a result, efficiently computing even tensor algebra expressions that only have a single operand requires specialized code for every possible tensor format.

Multiple tensor operands further complicate the issue as every combination of tensor formats for encoding the operands effectively requires specialized code as well. Different combinations of tensor formats require different strategies for merging the iteration spaces of the tensor operands, leading to compute code with dissimilar structures. Consider, for instance, the component-wise multiplication of two order-2 tensors (matrices) B and C. If

both matrices are stored as dense arrays, then they must share the same dense iteration space, in which case code to compute the operation can simply iterate over this shared iteration space (lines 1–2) and, at each point, multiply the corresponding components in the two tensor operands (line 3):

```
1 for (int i = 0; i < B1_size; ++i) {
2   for (int j = 0; j < B2_size; ++j) {
3     A[i * A2_size + j] = B[i * B2_size + j] * C[i * C2_size + j];
4   }
5 }
```

If B is stored in the CSR format, code to compute the operation can rely on the fact that dense arrays support efficient random access to just iterate the non-zero components of B (lines 1–3) and, for each non-zero coordinate, pick out the corresponding non-zero component from C (line 4). This method requires different code than before that iterates over the column dimension by reading from the `idx` array (lines 2–3):

```
1 for (int i = 0; i < B1_size; ++i) {
2   for (int pB2 = B2_pos[i]; pB2 < B2_pos[i + 1]; ++pB2) {
3     int j = idx[pB2];
4     A[i * A2_size + j] = B[pB2] * C[i * C2_size + j];
5   }
6 }
```

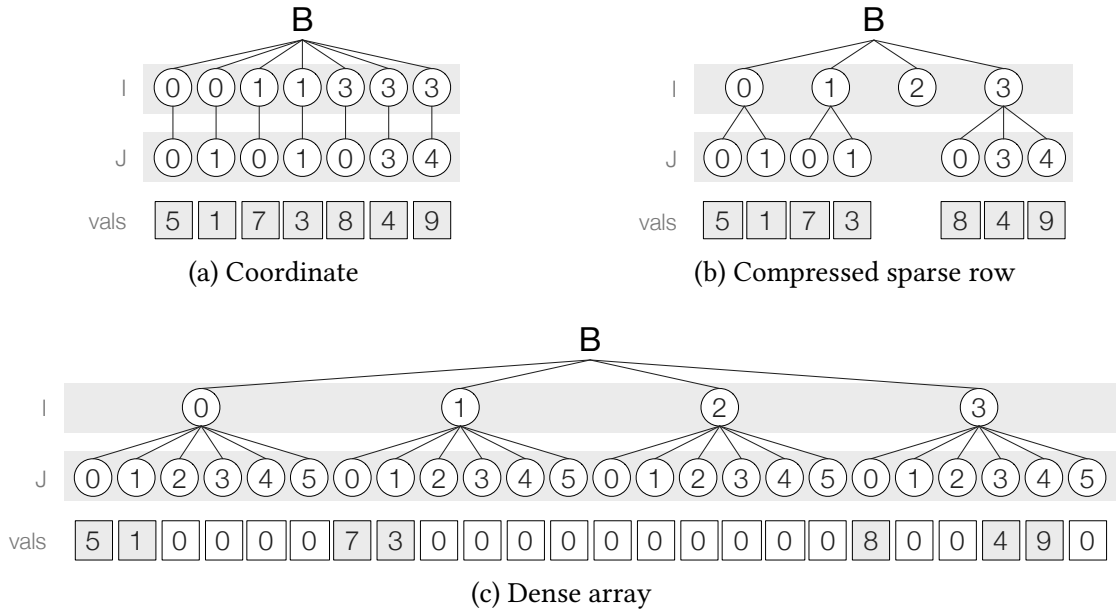
However, merging the iteration spaces of the tensor operands becomes much more complicated if C is also stored in the COO format (assuming non-zero coordinates are stored in order), as neither it nor the CSR format supports efficient random access. Thus, iterating over the column dimension requires much different code than before that instead co-iterates and merges the column dimensions of B and C (lines 7–18). Code to compute the operation also needs additional logic to ensure that the same row in C is not visited multiple times (lines 4–6), since the COO format may store duplicate copies of the same row coordinate (once for each non-zero in the row):

```
1 int pC1 = C1_pos[0];
2 while (pC1 < C1_pos[1]) {
```

```
  int i = C1_idx[pC1];
  int C1_segend = pC1 + 1;
  while (C1_segend < C1_pos[1] && C1_idx[C1_segend] == i)
    ++C1_segend;
  int pB2 = B2_pos[i];
  int pC2 = pC1;
  while (pB2 < B2_pos[i + 1] && pC2 < C1_segend) {
    int jB2 = B2_idx[pB2];
    int jC2 = C2_idx[pC2];
    int j = min(jB2, jC2);
    if (jB2 == j && jC2 == j) {
      A[i * A2_size + j] = B[pB2] * C[pB2];
    }
    if (jB2 == j) ++pB2;
    if (jC2 == j) ++pC2;
  }
  pC1 = C1_segend;
}
```

The sheer number of dissimilar kernels that are needed to effectively support a diverse range of tensor formats makes it impractical to implement all of them manually, motivating a metaprogramming approach that can instead generate these kernels automatically.

Of course, the combinatorial explosion of tensor format combinations also makes it unrealistic for any tensor algebra compiler to simply hard code for each individual format. This motivates the need for a common abstraction that can represent any tensor storage format, which a compiler can more manageably reason about and generate code for. In Chapter 3, we describe how tensor storage can be viewed as a hierarchy of levels that each encode coordinates along a tensor dimension; Figure 1-2 shows examples of such coordinate hierarchies for the same matrix stored in different formats. We then demonstrate how common variants of a wide range of tensor formats, including all of the ones mentioned above, can be represented as compositions of just six per-dimension formats that encode coordinate hierarchy levels: dense, range, compressed, singleton, offset, and hashed. We further show how these per-dimension level formats can be abstracted in



(a) Coordinate

(b) Compressed sparse row

(c) Dense array

Figure 1-2: Coordinate hierarchies for the same matrix from Figure 1-1a stored in different formats. The structure of each coordinate hierarchy reflects how the underlying storage format encodes non-zeros.

terms of five properties and five capabilities that describe how to iterate, index into, and modify a tensor dimension, which are exposed through a common abstract interface.

We then describe a code generation algorithm based on this abstraction that emits efficient code to compute any compound tensor algebra expression, where the operands can be stored in any combination of tensor formats that are assembled from level formats (Chapter 4). The algorithm works by emitting code that efficiently iterates and merges coordinate hierarchy representations of the tensor operands by calling functions that the level format interface exposes. A simple inlining pass then specializes the emitted code to work with specific tensor formats. The coordinate hierarchy abstraction enforces a strict separation between tensor storage formats and the code generation algorithm, ensuring that the algorithm does not have to directly reason about specific formats and thereby limiting the complexity of the algorithm, which keeps it maintainable.

To summarize, we make the following contributions:

Levelization We show how common variants of a whole host of tensor formats, including all of the ones mentioned above and many more (Chapter 2), can be represented as hierarchical compositions of just six per-dimension level formats (Figure 3-5).

Level abstraction We describe an abstraction for level formats (Chapter 3), which hides the specifics of how a level encodes a tensor dimension behind a common interface that exposes a level format's capabilities (e.g., how to iterate coordinates along a tensor dimension) and properties (e.g., whether coordinates are ordered).

Code generation We present a code generation technique that generates code to efficiently compute on sparse tensors stored in any format (Chapter 4), which reasons only about capabilities and properties of each dimension and does not rely on knowledge specific to particular level formats.

We implement our code generation technique as an extension to the open-source tensor algebra compiler `taco` [35]. `taco` is based on the work of Kjolstad et al., and its code generator is hard-coded to just two per-dimension storage types [34]. The modular nature of our extended compiler enables users to compute with a much wider set of tensor formats, which can moreover be expanded without needing to modify the code generation algorithm. We evaluate our technique against a range of existing sparse linear and tensor algebra libraries and find that our technique is able to generate sparse kernels for diverse storage formats with performance competitive with hand-implemented kernels (Chapter 5).

Chapter 2

Tensor Storage Formats

There exists a multitude of formats for storing tensors, many of which are commonly used in real-world applications. None of these formats is ideal in every circumstance; each can be useful depending on the structure of the data, the characteristics of the computation, and the underlying hardware. This makes it desirable to support efficiently computing on as many tensor formats as possible. As we will show, however, efficiently computing on different formats requires dissimilar code that specifically accommodates and exploits the idiosyncrasies of each format, which cannot be practically implemented by hand and is also challenging to automatically generate.

2.1 Existing Tensor Formats

Figures 2-1, 2-2, and 2-3 highlight examples of tensor formats that have been described in the literature. A straightforward way to store an nth-order tensor (i.e. a tensor consisting of n dimensions) is to use an n-dimensional dense array that explicitly stores all components of the tensor, including all the zeros. Figures 2-1b and 2-2b illustrate this for an order-1 tensor (i.e., a vector) and an order-2 tensor (i.e., a matrix) respectively. A desirable feature of dense arrays is that the value at any given coordinate can be accessed in constant time. However, storing a sparse tensor in a dense array is inefficient since a lot of memory is wasted explicitly storing zeros. All these zeros must also be processed when computing on the tensor even though they do not meaningfully contribute to the result, which is

detrimental to performance. For tensors with many dimensions that are large, it may even simply be impossible to use a dense array due to lack of memory; using a dense array to store the Amazon reviews tensor described in Chapter 1, for instance, would require 107 exabytes of storage (assuming each component is stored as a double-precision float).

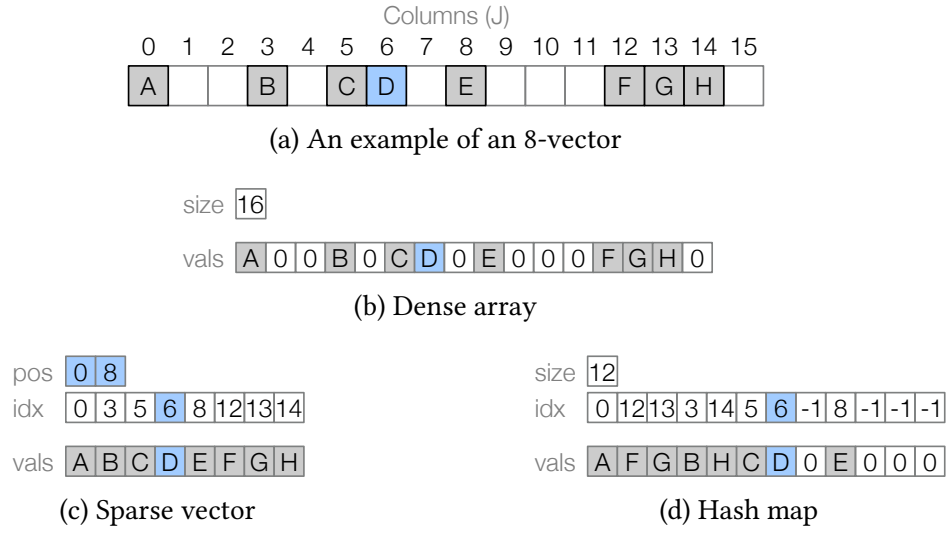
The simplest approach to efficiently store a sparse tensor is to just keep a list of its non-zero coordinates and values (Figures 2-1c, 2-2c, and 2-3b). This is sometimes referred to as sparse vectors in the context of 1st-order tensors and usually known as the coordinate (COO) format for tensors of higher order. Not only does the COO format consume only $O(\text{nnz})$ memory, it also closely resembles many common file formats for storing tensors including the Matrix Marketplace exchange format [50] and the FROSTT sparse tensor format [57]. This helps minimize preprocessing cost as inserting non-zero coordinates and values simply requires appending them to the `idx` and `vals` arrays.

Unlike dense arrays, the COO format does not provide constant-time random access. Hash maps (Figure 2-1d) eliminate this pitfall by using a hash table to store the tensor's non-zero coordinates. This, however, comes at the cost of losing the ability to efficiently iterate only non-zeros in order, which can complicate certain computations.

The compressed sparse row (CSR) format for sparse matrices (Figure 2-2d) addresses another drawback of the COO format, namely that it redundantly store coordinates. Redundant coordinates not only increases storage cost but also reduces the performance of memory bandwidth-bound computations like sparse matrix-vector multiplication (SpMV). In Figure 2-2c, for instance, the row coordinates of the first three non-zero components are all explicitly stored even though they actually belong to the same row of the matrix. The CSR format compresses out these redundant row coordinates using an auxiliary array (`pos` in Figure 2-2d, though it is also often labeled `row_ptr` in other sources) that keeps track of which segment of non-zeros belongs to which row. The compressed sparse column (CSC) format (Figure 2-2e) follows the same basic principle but instead compresses out redundant column coordinates. The doubly compressed sparse row (DCSR) format (Figure 2-2f) and the corresponding doubly compressed sparse column (DCSC) format, proposed by Buluç and Gilbert, achieve additional compression for hypersparse matrices by only keeping track of the rows or columns that actually contain non-zeros [16]. For higher-order ten-

sors, Smith and Karypis described a generalization of (D)CSR, which they refer to as the compressed sparse fibers (CSF) format (Figure 2-3c), that uses the same index structures as CSR to represent each dimension and store only the non-zero fibers of a tensor [58, 59]. However, all these compressed formats tend to be much more complicated and expensive to assemble than the COO format and even harder to modify.

Many important classes of applications work with tensors whose non-zero components are distributed in some sort of regular pattern. Matrices that encode vertex-edge connectivity of unstructured meshes, for instance, tend to have a fixed number of non-zero components per row. The ELLPACK (ELL) format (Figure 2-2g) takes advantage of this by storing the column coordinates and values of all non-zero components such that the *k*th non-zeros of all rows are kept contiguous in memory, which makes it possible to vectorize SpMV [22]. If all the non-zero components are restricted to a few densely-filled diagonals, then the coordinates of the non-zeros can actually just be computed from the offsets of the diagonals. which allows the diagonal (DIA) format (Figure 2-2h) to forgo explicitly storing the column coordinates altogether. For matrices that do not conform to the assumed structures though, these structured formats must unnecessarily store many zeros, which can drastically increase storage cost and degrade computational performance.



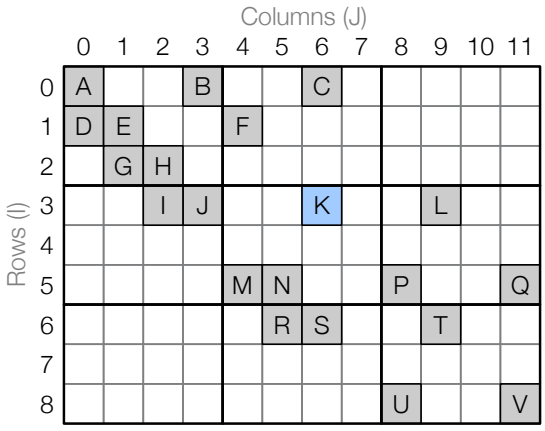
(a) An example of an 8-vector

(b) Dense array

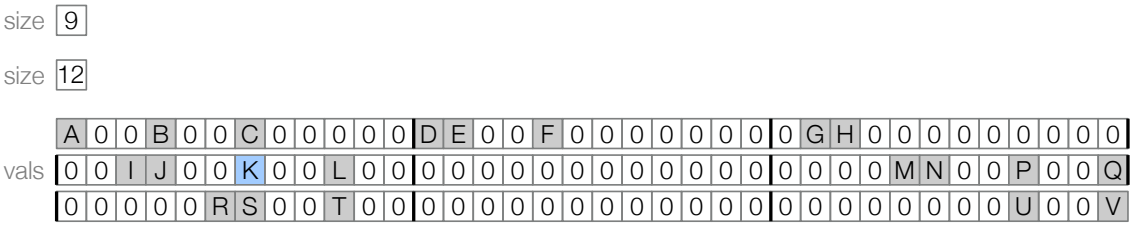
(c) Sparse vector

(d) Hash map

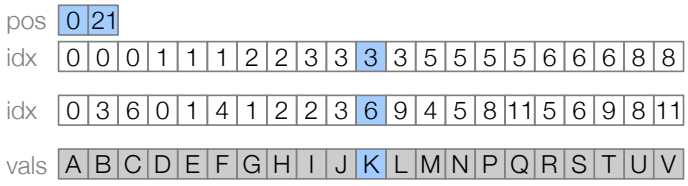
Figure 2-1: Examples of various vector storage formats storing the same vector. Cells shaded gray identify non-zero components in the vector while cells shaded light blue identify elements in the physical indices that encode a particular non-zero component.



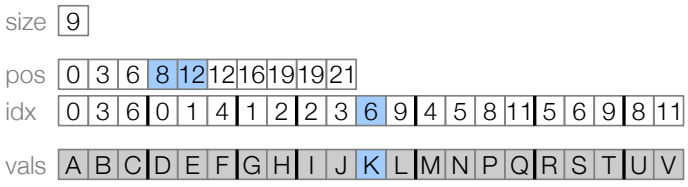
(a) An example of a 9×12 matrix



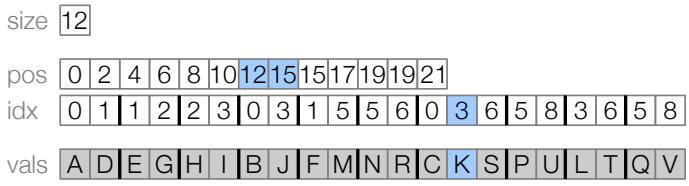
(b) Dense array



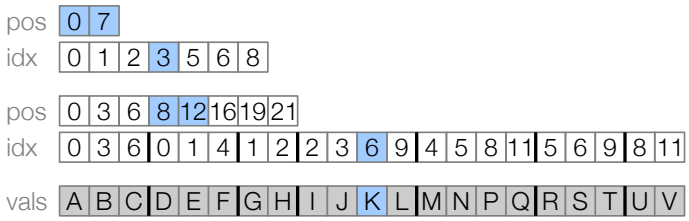
(c) Coordinate



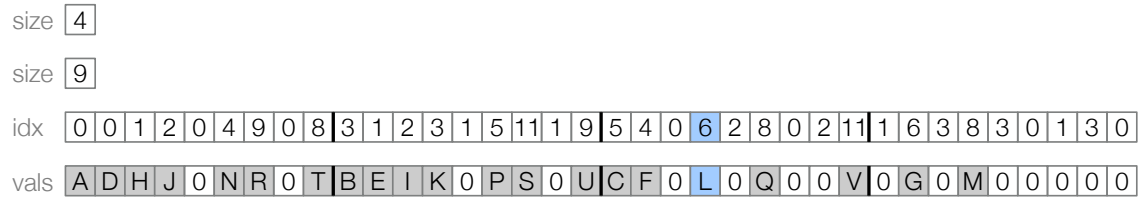
(d) Compressed sparse row



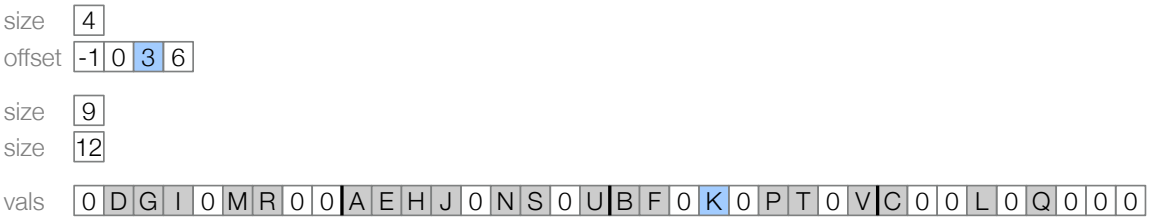
(e) Compressed sparse column



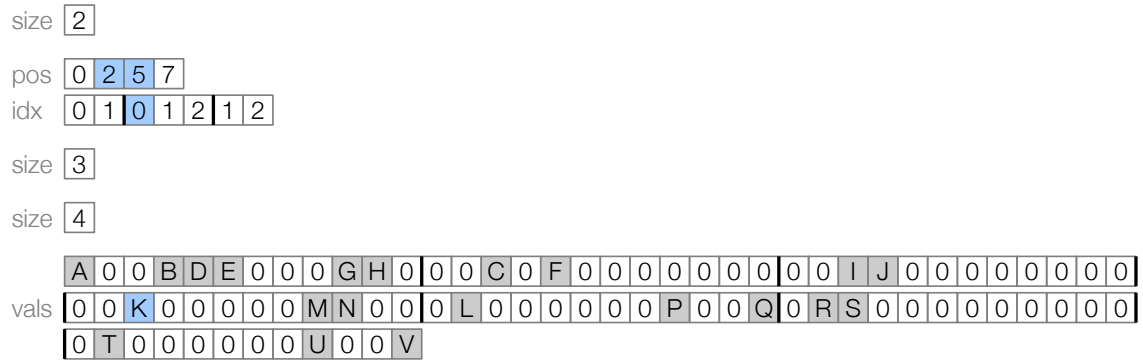
(f) Doubly compressed sparse row



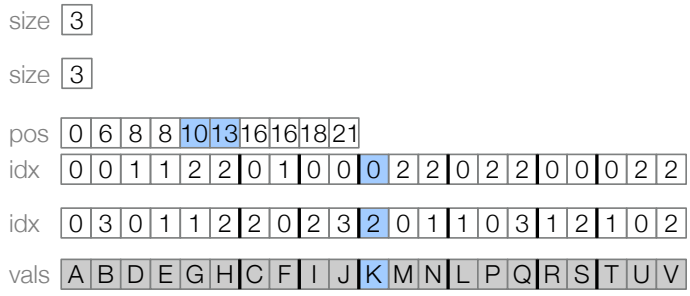
(g) ELLPACK



(h) Diagonal

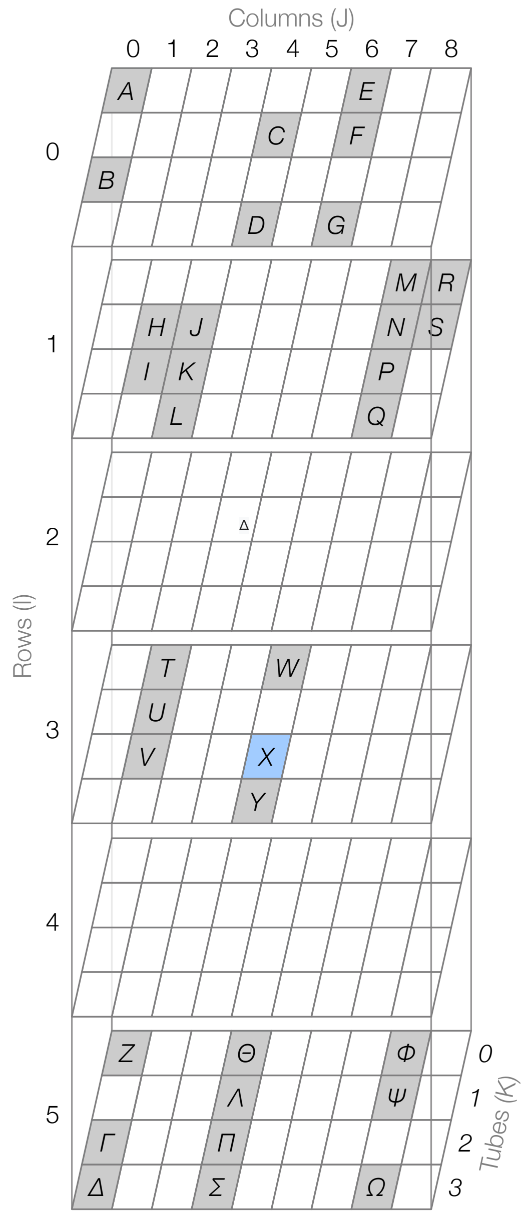


(i) Block compressed sparse row



(j) Compressed sparse block

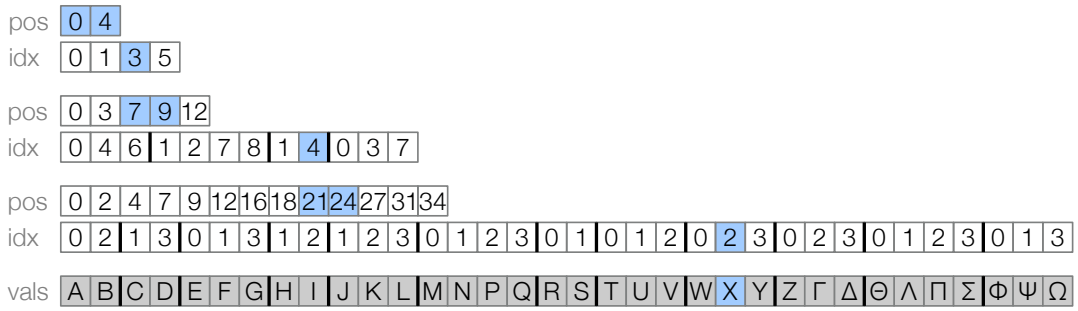
Figure 2-2: Examples of various matrix storage formats storing the same matrix.



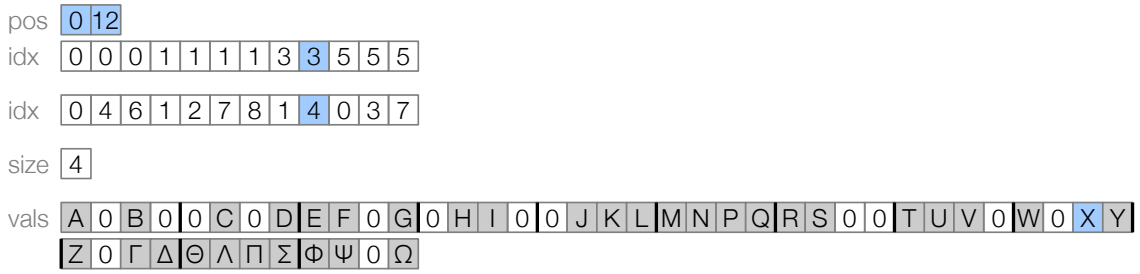
(a) An example of a 6×9×4 tensor

pos	0	34																																
idx	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	5	5	5	5	5	5	5	5	5	5
idx	0	0	4	4	6	6	6	1	1	2	2	2	7	7	7	7	8	8	1	1	1	4	4	4	0	0	0	3	3	3	3	7	7	7
idx	0	2	1	3	0	1	3	1	2	1	2	3	0	1	2	3	0	1	0	1	2	0	2	3	0	2	3	0	1	2	3	0	1	3
vals	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	R	S	T	U	V	W	X	Y	Z	Γ	Δ	Θ	Λ	Π	Σ	Φ	Ψ	Ω

(b) Coordinate



(c) Compressed sparse fiber



(d) Mode-generic sparse tensor

Figure 2-3: Examples of various tensor storage formats storing the same 3rd-order tensor.

The block compressed sparse row (BCSR) format (Figure 2-2i) generalizes CSR by storing a dense *block* of non-zeros in the `vals` array at every non-zero coordinate. This exposes opportunities for vectorization and vector reuse in SpMV and is ideal for blocked matrices from FEM applications. The mode-generic sparse tensor storage format (Figure 2-3d), proposed by Baskaran et al., generalizes the idea of the BCSR format to higher-order tensors by storing a tensor as a sparse collection of arbitrary-order dense blocks, using the COO format (the `idx` arrays in Figure 2-3d) to represent the sparse collection [10]. By contrast, the compressed sparse block (CSB) format (Figure 2-2j), proposed by Buluç et al., represents a matrix as a dense collection of sparse blocks stored in the COO format (the `idx` arrays in Figure 2-2j) and can be viewed as the converse of BCSR [15].

2.2 Supporting Diverse Formats

Maximizing the performance of sparse tensor computations with each of these different formats requires specialized code that accommodates and exploits each format's idiosyncrasies. As the examples below demonstrate though, even code that does something as

seemingly simple as iterating a tensor can vary significantly in structure from one format to another, motivating a metaprogramming approach that removes the need for programmers to manually implement all these wildly different kernels. For instance, iterating a dense array that stores a vector entails looping over *coordinates* along the vector's dimension and using those coordinates to index into the `vals` array and retrieve the corresponding component values, as Figure 2-4a illustrates. The code below implements this algorithm:

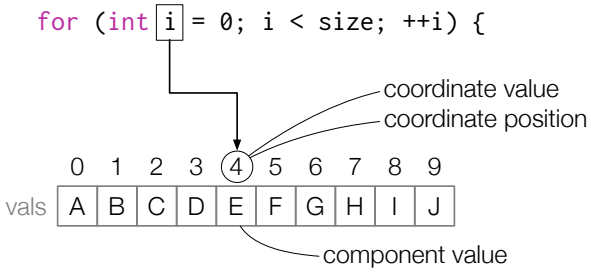
```
1 for (int i = 0; i < size; ++i) {
2   double v = vals[i];
3   printf("x(%d) = %f\n", i, v);
4 }
```

Iterating a sparse vector, on the other hand, requires looping over *positions* in the physical indices and accessing the `idx` and `vals` arrays at each position to retrieve the corresponding coordinate and component value, as Figure 2-4b demonstrates. The code below, which is significantly different from the one shown before, implements this algorithm:

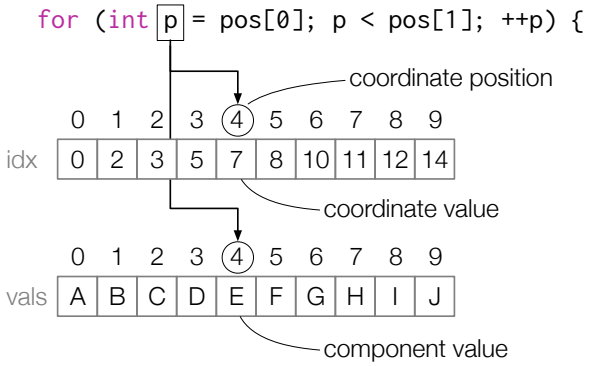
```
1 for (int p = pos[0]; p < pos[1]; ++p) {
2   int i = idx[p];
3   double v = vals[p];
4   printf("x(%d) = %f\n", i, v);
5 }
```

Kernels for iterating many other common tensor formats mix elements of both examples above. Efficiently iterating the CSR format, for instance, involves looping over coordinates along the row dimension (line 1) and, for each row coordinate, iterating over positions in the `idx` and `vals` arrays that store the corresponding non-zero columns (line 2):

```
1 for (int i = 0; i < size; ++i) {
2   for (int p = pos[i]; p < pos[i + 1]; ++p) {
3     int j = idx[p];
4     double v = vals[p];
5     printf("A(%d, %d) = %f\n", i, j, v);
6   }
7 }
```



(a) Iterating a dense array



(b) Iterating a sparse vector

Figure 2-4: Iterating a dense array and a sparse vector requires accessing their corresponding physical indices in very different ways. The former entails looping over coordinates (`i`) and indexing into the `vals` array using those coordinates, while the latter entails looping over positions (`p`) and accessing the coordinates and component values stored in the `idx` and `vals` arrays at those positions.

Critically, the code above iterates over the row dimension in the outer loop and the column dimension in the inner loop, since the CSR format allows efficiently determining which columns encoded in the `idx` array correspond to a particular row by simply indexing the `pos` array. To iterate over the column dimension in the outer loop, on the other hand, would require scanning the entire index structure for each column coordinate in order to identify all the non-zero rows in that column, which is clearly inefficient. The reverse is true for the CSC format; to efficiently iterate a CSC matrix requires iterating over the column dimension in the outer loop and the row dimension in the inner loop (but with otherwise identical code as before):

```
1 for (int j = 0; j < size; ++j) {
2   for (int p = pos[j]; p < pos[j + 1]; ++p) {
3     int i = idx[p];
4     double v = vals[p];
5     printf("A(%d, %d) = %f\n", i, j, v);
6   }
7 }
```

In all of the examples above, coordinates along different dimensions of a tensor can be iterated independently using one iterator per dimension. However, certain tensor formats require iterating multiple dimensions concurrently when enumerating non-zeros. Iterating a COO tensor, for instance, is typically accomplished with a single non-nested loop (line 1) and a shared iteration variable (`p`) that is used to directly index into all of the `idx` arrays and load the corresponding coordinates for all dimensions (lines 2–4):

```
1 for (int p = pos[0]; p < pos[1]; ++p) {
2   int i = idx1[p];
3   int j = idx2[p];
4   int k = idx3[p];
5   double v = vals[p];
6   printf("A(%d, %d, %d) = %f\n", i, j, k, v);
7 }
```

For some formats, the coupling between tensor dimensions extends beyond iterators to the actual coordinates. Efficiently iterating a DIA matrix, for example, requires looping over

coordinates along the row dimension (line 2) and, for each row coordinate, computing the corresponding column coordinate as a function (more specifically, an additive offset) of the row coordinate (line 3):

```
1 for (int d = 0; d < size1; ++d) {
2   for (int i = max(0, -offset[d]), i < min(size2, size3 - offset[d]); ++i) {
3     int j = i + offset[d];
4     double v = vals[d * size2 + i];
5     printf("A(%d, %d) = %f\n", i, j, v);
6   }
7 }
```

Additionally, some tensor formats naturally impose constraints on how coordinates can be enumerated efficiently. As an example, the positions of coordinates in a hash map sparse vector's `idx` array are determined by the coordinates' hash values. Consequently, no code that iterates a hash map sparse vector's non-zeros in asymptotically optimal time, such as the one shown below, can guarantee that coordinates are enumerated in order.

```
1 for (int p = pos[0]; p < pos[1]; ++p) {
2   if (idx[p] != -1) {
3     int i = idx[p];
4     double v = vals[p];
5     printf("x(%d) = %f\n", i, v);
6   }
7 }
```

To efficiently iterate a hash map sparse vector in order requires significantly more complex code. One approach, shown below, is to first assemble a scratch array that stores all non-zero coordinates from the hash map in sorted order (lines 4–12), and to then iterate the scratch array instead of the hash map (lines 14–18). As with any method that guarantees ordered enumeration for a hash map sparse vector, this code exhibits worse performance than the previous example due to the need for the sort operation and must therefore be avoided if a computation does not actually need in-order enumeration.

```
1 typedef struct { int i; int p; } coord;
```

```
2  coord coords[];
3
4  int nnz = 0;
5  for (int p = pos[0]; p < pos[1]; ++p) {
6    if (idx[p] != -1) {
7      coords[nnz].p = p;
8      coords[nnz].i = idx[p];
9      ++nnz;
10   }
11 }
12 sort(coords, nnz);
13
14 for (int n = 0; n < nnz; ++n) {
15   int i = coords[n].i;
16   double v = vals[coords[n].p];
17   printf("x(%d) = %f\n", i, v);
18 }
```

These are just a few examples of the kinds of code that any compiler for sparse tensor computation has to be able to generate in order to achieve performance that is competitive with hand-implemented code for a diverse range of tensor storage formats.

Chapter 3

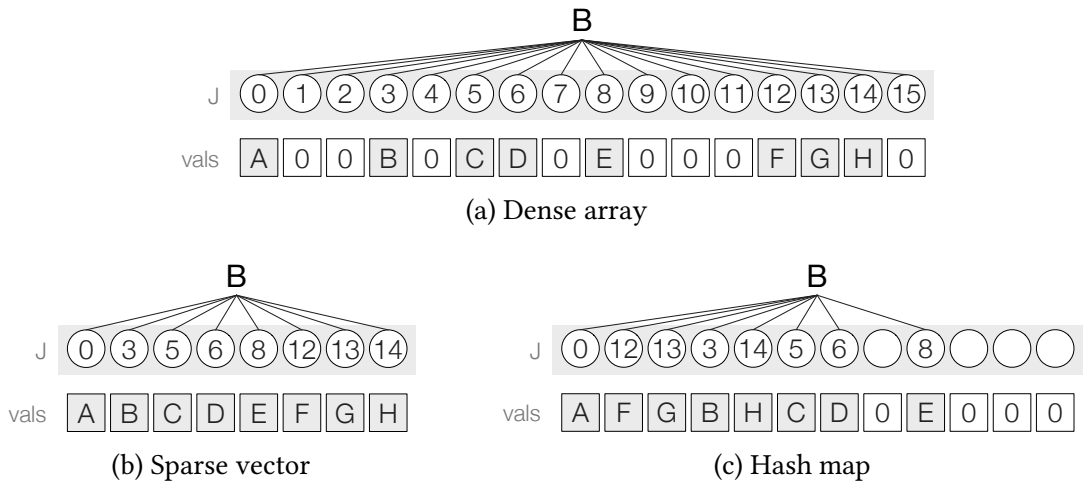
Tensor Storage Abstraction

The tensor storage formats examined in the previous chapter represent just a subset of the many formats that have been described in the literature. Moreover, most computations deal with multiple inputs that can each be stored in a different format, and, as we discussed in Chapter 1, efficiently computing with any particular combination of formats requires specialized code with distinct structure. Given the sheer number of tensor formats and an even larger number of combinations of these formats, it is impractical for any tensor algebra compiler to hard code for each individual format. This motivates the need for a common abstraction that can represent any tensor storage format, which a code generator can more manageably reason about and work with.

As we demonstrate in the rest of this chapter, common variants of all the aforementioned tensor formats can, in fact, be described in terms of just six per-dimension formats that can be composed in a hierarchical fashion. We show how per-dimension formats such as these can be abstracted in terms of their properties and capabilities, which generalize recurring patterns that we examined in Section 2.2 for efficiently accessing a tensor and which guide the format-agnostic code generation technique we will describe in Chapter 4.

3.1 Coordinate Hierarchies

The idea of per-dimension formats can best be understood by viewing tensor storage as a composition of *levels* that each encode coordinates along one tensor dimension. Edges



(a) Dense array

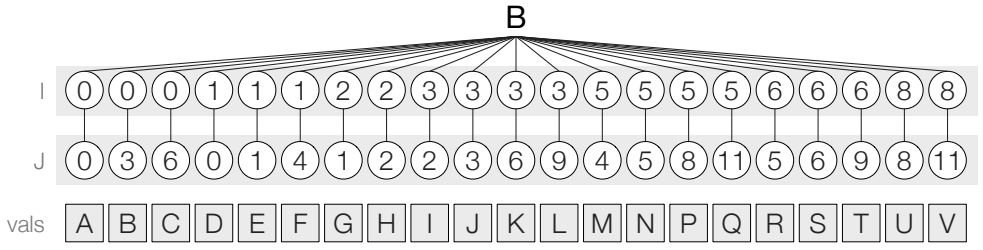
(b) Sparse vector

(c) Hash map

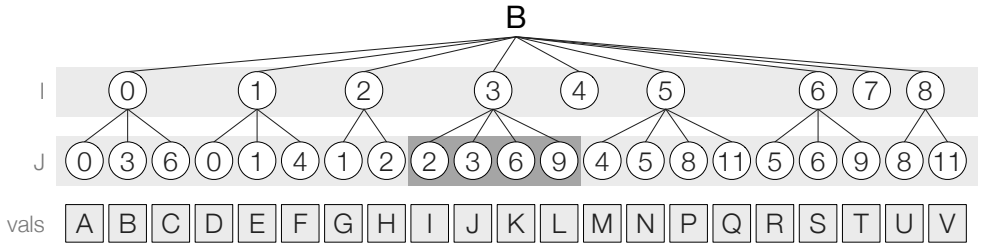
Figure 3-1: Coordinate hierarchies for the same vector from Figure 2-1a stored in different vector formats. A coordinate hierarchy's structure reflects how the underlying storage format encodes non-zeros; the arrangement of coordinates in (b) and (c), for instance, mirrors the order in which coordinates are stored in the `idx` array.

connect coordinates in adjacent levels, forming a *coordinate hierarchy*. Figures 3-1, 3-2, 3-3, and 3-4 show some examples of coordinate hierarchies for the same vector and matrix (from Figure 2-1a and Figure 2-2a) stored in different formats, with component values at the bottom of each hierarchy. A fully labeled path from the root to a leaf in a coordinate hierarchy describes one tensor component, with its coordinates specified by the labels along the path. In Figure 3-2b, for instance, the rightmost path represents the tensor component $B(8, 11)$ with its corresponding value V. As we will show in Chapter 4, representing tensor storage in this hierarchical fashion enables our code generation technique to decompose arbitrary computations into simpler problems of merging levels that represent tensor dimensions, which makes the problem of generating tensor algebra kernels tractable.

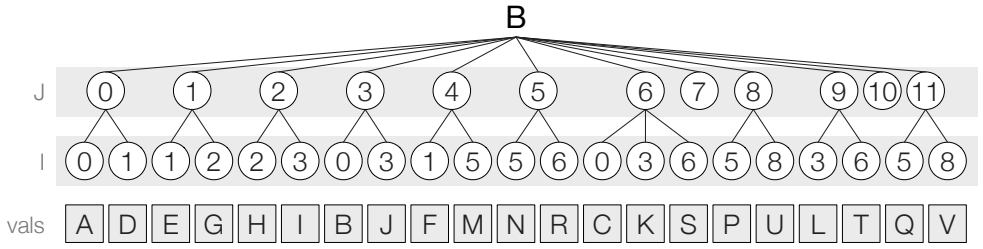
The structure of a tensor's coordinate hierarchy reflects how the underlying storage format encodes the tensor's non-zeros. For example, the coordinate hierarchy for a tensor stored as a dense array (Figure 3-1a) forms a full tree with every tensor component represented by a path, which mirrors how a dense array stores every value including zeros. On the other hand, the coordinate hierarchy for a COO tensor (Figure 3-2a) consists of chains of coordinates that each encodes a non-zero tensor component, which reflects how the COO format explicitly stores the complete tensor coordinate of every non-zero. By



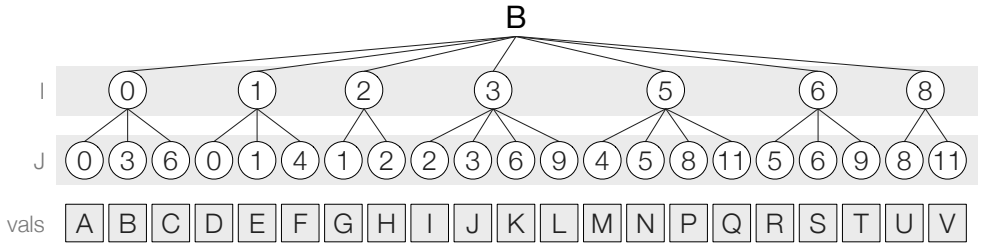
(a) Coordinate



(b) Compressed sparse row

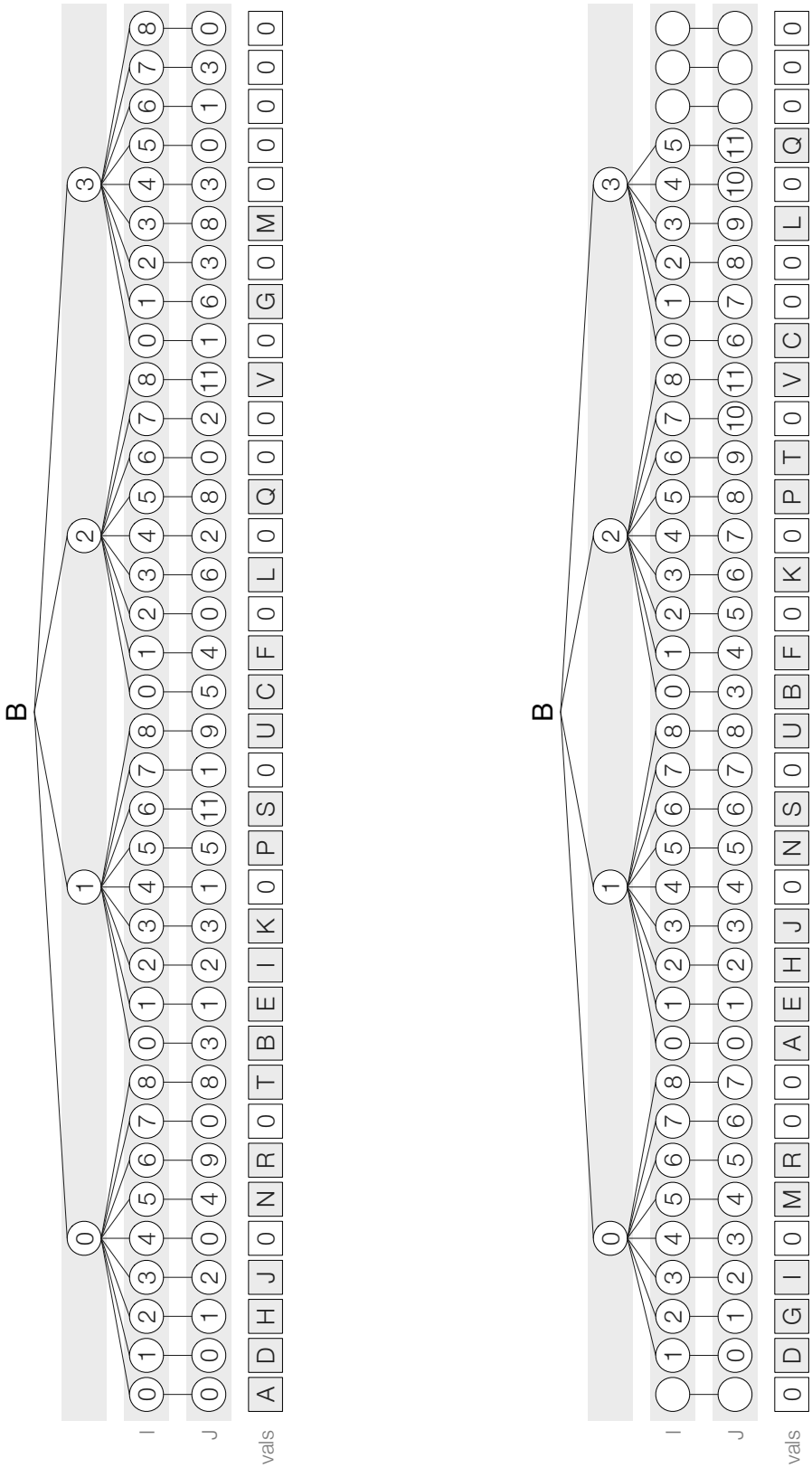


(c) Compressed sparse column



(d) Doubly compressed sparse row

Figure 3-2: Coordinate hierarchies for the same matrix from Figure 2-2a stored in different matrix formats. The label beside each level identifies the matrix dimension it represents.



(a) ELLPACK (b) Diagonal

Figure 3-3: Coordinate hierarchies for the same matrix stored in the ELL and DIA formats.

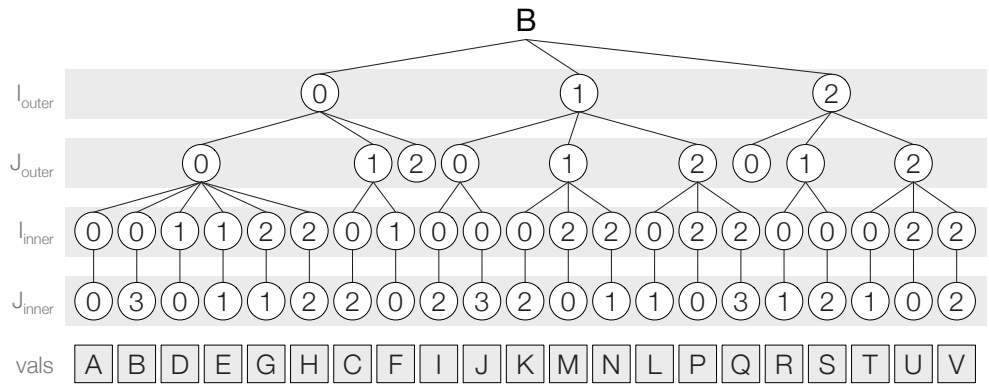


Figure 3-4: Coordinate hierarchy for the same matrix stored in the CSB format.

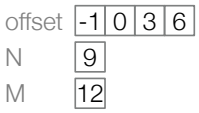
contrast, the coordinate hierarchy for a CSR matrix (Figure 3-2b) has column coordinates that belong to the same row share the same parent, which reflects how the CSR format compresses out redundant row coordinates using the auxiliary `pos` array.

A coordinate hierarchy has one level—shaded light gray in Figure 3-2—for every tensor dimension, and per-dimension level formats describe how to store the hierarchy levels. Each position (a node) in a coordinate hierarchy level may encode some coordinate (the number in the node) along the corresponding tensor dimension. (Alternatively, a position may contain a dummy node that does not encode any coordinate, like the empty nodes in Figure 3-1c.) Each position may also be connected to a parent in the previous level. Coordinates that share the same parent are referred to as *siblings*; the coordinates highlighted in dark gray in Figure 3-2b, for instance, are siblings that share the same parent coordinate encoding the fourth row of the matrix (we assume coordinates are zero-based).

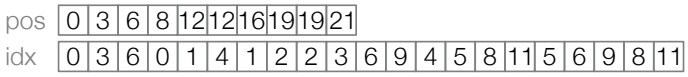
A level's format describes some encoding of a list of numbers that represent coordinates along the corresponding dimension. Depending on the level format, such a list of dimension coordinates can be implicitly encoded (e.g., as a range of coordinates succinctly described by its bounds) or explicitly stored (e.g., as a segmented vector). Here we propose six distinct level formats that are sufficient to represent all the tensor formats described in Chapter 2. Given a parent coordinate in any coordinate hierarchy, these level formats encode the corresponding child coordinates as follows:

Dense levels store the size of the corresponding tensor dimension (N) and encode all coordinates in the range $[0, N)$. Figure 3-2b shows the row dimension of a CSR matrix encoded as a dense level with the following data structure:

Range levels encode all coordinates in a range with bounds computed as a function of a tensor diagonal offset stored in the `offset` array and tensor dimension sizes N and M. Figure 3-3b shows the row dimensions of a DIA matrix encoded as a range level with the following data structures:

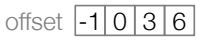


Compressed levels store coordinates in a segment of a segmented vector (`idx`), with the bounds of the segment stored in the `pos` array. Figure 3-2b shows the column dimension of a CSR matrix encoded as a compressed level with the following data structures:

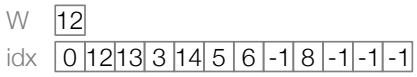


Singleton levels store a single coordinate without any siblings in the `idx` array. Figure 3-2a shows the column dimension of a COO matrix encoded as a singleton level with the following data structure:

Offset levels encode a single coordinate without any siblings, computed from the parent coordinate with an offset stored in the `offset` array. Figure 3-3b shows the column dimension of a DIA matrix encoded as an offset level with the following data structure:



Hashed levels store coordinates in a hash map (`idx`) of size W. Figure 3-1c shows a hash map vector encoded as a hashed level with the following data structures:

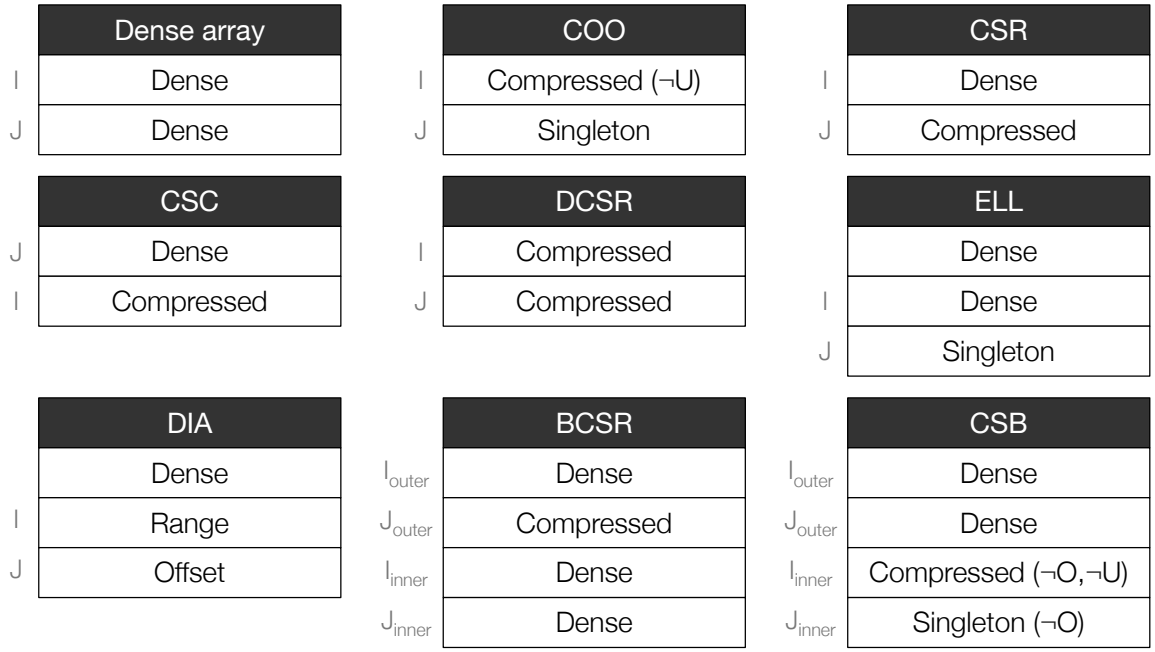


Section 3.2 documents the precise semantics of the level formats above, and Figure 3-5 shows concretely how these level formats can be composed to construct all of the tensor formats described in Chapter 2. We cast structured matrix formats, like BCSR, as formats for higher-order tensors; the added dimensions expose more complex matrix structure that require component values to be stored in non-lexicographic order with respect to the row and column coordinates.

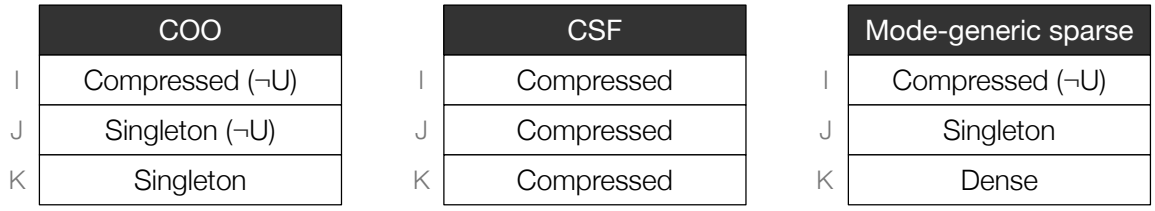
That only six level formats are needed to represent so many tensor formats reflects how some formats are, in essence, compositions of other formats. The mode-generic sparse tensor format, for instance, can be viewed as a composition of the COO format with the dense array format, where scalar component values in the COO format (represented by the compressed and singleton level types) are replaced by dense arrays (represented by the dense level type). Additionally, a format may use the same type of physical index to encode multiple dimensions, thus enabling level format reuse. For example, the COO format encodes each tensor dimension with one `idx` array that stores coordinates along the dimension, so a COO tensor of any order n can be represented by a hierarchy with one compressed level (which also stores the number of non-zeros) and $n - 1$ singleton levels.

The code generation algorithm described in Chapter 4 accesses and modifies coordinate hierarchy levels through an abstract interface, which exposes common patterns for manipulating physical indices in tensor storage. This approach ensures that the algorithm is not specialized—and thereby tied—to specific formats, which makes the compiler more extensible and maintainable since adding support for more formats does not require wholesale changes to the code generation mechanism. The abstract interface to a coordinate hierarchy level consists of level capabilities and properties. Level capabilities, which are described in more detail in Section 3.2 and Section 3.4, instruct the compiler how to correctly index into or iterate over coordinates encoded by a level and how to add coordinates

(a) Vector storage formats



(b) Matrix storage formats



(c) Tensor storage formats

Figure 3-5: Common variants of tensor formats described in Chapter 2, represented as hierarchical compositions of per-dimension level formats. We cast structured matrix formats as higher-order tensor formats. The label beside a level identifies the tensor dimension it represents. Unless otherwise stated, all levels other than hashed levels are assumed to ordered and unique (see Section 3.3); hashed levels are assumed to be unordered and unique. (¬O) denotes an unordered level and (¬U) denotes a non-unique level.

to a level. Properties of a level, which are described in Section 3.3, let the compiler emit optimized code that exploits tensor attributes to increase computational performance.

3.2 Level Capabilities

Every coordinate hierarchy level must provide a set of *capabilities* that can be used to access or modify its coordinates. Each capability is defined in terms of *level functions* that a level must implement in order to support the capability, and which provide an abstraction for manipulating physical indices in tensor storage in a format-agnostic manner.

For instance, column dimensions in CSR matrices are represented by compressed levels (Figure 3-5), which provide the coordinate position iteration capability (Table 3.1). As Table 3.1 also shows, the coordinate position iteration capability is defined by two level functions: `pos_iter` and `pos_access`. Thus, to access the column coordinates highlighted in dark gray in Figure 3-2b, one can first call `pos_iter` with the position of row coordinate (3) as input to identify the positions of the column coordinates, and then call `pos_access` with each position as input to determine the column coordinate values. Under the hood, `pos_iter` indexes the `pos` array to locate the segment in the `idx` array that stores the column coordinates, while `pos_access` retrieves each of the column coordinates from `idx`. These level functions describe exactly how CSR indices can be efficiently accessed while effectively hiding details such as the existence of the `pos` and `idx` arrays from the caller.

We define similar level functions for each of the five other types of hierarchy levels described in the previous section; Table 3.1 lists the access capabilities that are defined for each level type along with definitions of the corresponding level functions. Our code generation algorithm emits code that calls level functions in order to access physical indices, which ensures that the algorithm is not tied to any specific type of physical index.

The rest of this section describes three different access capabilities: *coordinate value iteration*, *coordinate position iteration*, and *locate*. These capabilities generalize the various recurring patterns we saw in Section 2.2 for accessing physical indices in tensor storage. Every coordinate hierarchy level must provide coordinate value iteration or coordinate

Table 3.1: Access capabilities that are supported by each of the six level types listed in Section 3.1, and the definitions of their corresponding level functions.

Level Type	Supported Capability	Level Function Definitions

Dense	Coord. value iteration	`coord_iter(`i_1`, ..., `i_{k-1}`):`
		`  return <0, `N_k`>`
		
		`coord_access(`p_{k-1}`, `i_1`, ..., `i_k`):`
		`  return <`p_{k-1}` * `N_k` + `i_k`, true>`

	Locate	`locate(`p_{k-1}`, `i_1`, ..., `i_k`):` `  return <`p_{k-1}` * `N_k` + `i_k`, true>`

Range	Coord. value iteration	`coord_iter(`i_1`, ..., `i_{k-1}`):`
		`  return <max(0, -offset[`i_{k-1}`]),`
		`  min(`N_k`, `M_k` - offset[`i_{k-1}`])>`
		
		`coord_access(`p_{k-1}`, `i_1`, ..., `i_k`):`
		`  return <`p_{k-1}` * `N_k` + `i_k`, true>`

Compressed	Coord. position iteration	`pos_iter(`p_{k-1}`):`
		`  return <pos[`p_{k-1}`], pos[`p_{k-1}` + 1]>`
		
		`pos_access(`p_k`, `i_1`, ..., `i_{k-1}`):`
		`  return <idx[`p_k`], true>`

Singleton	Coord. position iteration	`pos_iter(`p_{k-1}`):`
		`  return <`p_{k-1}`, `p_{k-1}` + 1>`
		
		`pos_access(`p_k`, `i_1`, ..., `i_{k-1}`):`
		`  return <idx[`p_k`], true>`

Offset	Coord. position iteration	`pos_iter(`p_{k-1}`):`
		`  return <`p_{k-1}`, `p_{k-1}` + 1>`
		
		`pos_access(`p_k`, `i_1`, ..., `i_{k-1}`):`
		`  return <`i_{k-1}` + offset[`i_{k-2}`], true>`

Hashed	Coord. position iteration	`pos_iter(`p_{k-1}`):`
		`  return <`p_{k-1}` * `W_k`, (`p_{k-1}` + 1) * `W_k`>`
		
		`pos_access(`p_k`, `i_1`, ..., `i_{k-1}`):`
		`  return <idx[`p_k`], idx[`p_k`] != -1>`

	Locate	`locate(`p_{k-1}`, `i_1`, ..., `i_k`):` `  int `p_k` = `i_k` % `W_k` + `p_{k-1}` * `W_k` ` if (idx[`$p_k$`] != `$i_k$` && idx[`$p_k$`] != -1) {` ` int end = `$p_k$` `  do {` `  `p_k` = (`p_k` + 1) % `W_k` + `p_{k-1}` * `W_k` ` } while (idx[`$p_k$`] != `$i_k$` &&` ` idx[`$p_k$`] != -1 && `$p_k$` != end)` ` }` ` return <`$p_k$`, idx[`$p_k$`] == `$i_k$`>`

position iteration and can optionally provide the locate capability. Section 3.4 describes assembly capabilities for modifying hierarchy levels.

Coordinate Value Iteration Capability The coordinate value iteration capability enables iteration over a set of coordinates that are identified by their values, generalizing the method shown in Figure 2-4a for iterating a dense array. It is suited for physical indices that succinctly encode the set of non-zero coordinates along a tensor dimension.

Coordinate value iteration is defined in terms of two level functions, one that returns an iterator over coordinates along a tensor dimension (`coord_iter`) and one that accesses each coordinate's position within the corresponding coordinate hierarchy level (`coord_access`):

```
coord_iter(i_1, ..., i_{k-1}) -> <i_k_begin, i_k_end>
coord_access(p_{k-1}, i_1, ..., i_k) -> <p_k, found>
```

More precisely, given a list of ancestor coordinates ($i_1 \ldots i_{k-1}$), the function `coord_iter` returns the bounds of an iterator into potential coordinates (i_k) that has those ancestors. For each coordinate (i_k), the function `coord_access` returns the position of a child of p_{k-1} that encodes (i_k), or returns `found` as false if the coordinate does not actually exist. These functions can be used to iterate tensor coordinates in the general case as follows:

```
i_k_begin, i_k_end = coord_iter(i_1, ..., i_{k-1});
for (i_k = i_k_begin; i_k < i_k_end; ++i_k) {
  p_k, found = coord_access(p_{k-1}, i_1, ..., i_k);
  if (found) {
    // coordinates and values dominated by i_k at position
    // p_k encode the subtensor B(i_1, ..., i_k, :, ..., :)
  }
}
```

In practice, many level formats that provide the coordinate value iteration capability (including dense and range level types) will always return `found` as true, in which case the code above can be straightforwardly optimized by removing the conditional:

```
i_k_begin, i_k_end = coord_iter(i_1, ..., i_{k-1});
for (i_k = i_k_begin; i_k < i_k_end; ++i_k) {
```

```
  p_k, _ = coord_access(p_{k-1}, i_1, ..., i_k);
  // coordinates and values dominated by i_k at position
  // p_k encode the subtensor B(i_1, ..., i_k, :, ..., :)
}
```

Coordinate Position Iteration Capability The coordinate position iteration capability, on the other hand, enables iteration over a set of coordinates that are identified by their positions, generalizing the method shown in Figure 2-4b for iterating a sparse vector. It is suited for physical indices that explicitly store coordinates or that encode coordinates along one tensor dimension as a function of coordinates along other dimensions.

Coordinate position iteration is also defined in terms of two level functions, one that returns an iterator over positions within a coordinate hierarchy level (`pos_iter`) and one that accesses the coordinate at each position (`pos_access`):

```
pos_iter(p_{k-1}) -> <p_k_begin, p_k_end>
pos_access(p_k, i_1, ..., i_{k-1}) -> <i_k, found>
```

More precisely, given a coordinate at position p_{k-1}, the function `pos_iter` returns the bounds of an iterator into potential positions p_k with p_{k-1} as the parent. For each position p_k, the function `pos_access` returns the coordinate encoded at that position, or returns `found` as false if the coordinate either does not exist or is not actually a child of p_{k-1}. These functions can be used to iterate tensor coordinates in the general case as follows:

```
p_k_begin, p_k_end = pos_iter(p_{k-1});
for (p_k = p_k_begin; p_k < p_k_end; ++p_k) {
  i_k, found = pos_access(p_k, i_1, ..., i_{k-1});
  if (found) {
    // coordinates and values dominated by i_k at position
    // p_k encode the subtensor B(i_1, ..., i_k, :, ..., :)
  }
}
```

This code is similar in structure to the one shown previously for coordinate value iteration, but with the roles of i_k and p_k reversed. And, as is the case with coordinate value

iteration, many level formats that provide the coordinate position iteration capability (e.g., compressed level types) will in practice always return `found` as true, in which case the code above can be optimized by removing the conditional.

Locate Capability The locate capability provides random access into a coordinate hierarchy level through function that computes the position of a coordinate:

`locate(`$\mathtt{p}_{k-1}$`, `$\mathtt{i}_1$`, ..., `$\mathtt{i}_k$`) -> <`$\mathtt{p}_k$`, found>`

The `locate` function has similar semantics as `coord_access`. Given a coordinate (i_{k-1})—with ancestor coordinates ($i_1 \ldots i_{k-2}$)—at position p_{k-1} in the previous level, `locate` queries whether the coordinate at p_{k-1} has a child that encodes the coordinate (i_k). If so, then `locate` returns the position of that child and returns `found` as true; otherwise it returns `found` as false. Traversing a single path in a coordinate hierarchy—in other words, accessing a single tensor component—can be accomplished by successively calling `locate` at every level, assuming they all support the locate capability.

The locate capability is useful in tensor contraction kernels that multiply tensors. As we will see in Chapter 4, if at least one of the operands supports the locate capability, then the generated code need only iterate over the operands that do not support the locate capability; values from the operands that do support the capability can be gathered with the `locate` function, which our technique assumes will execute in constant time. Thus, the cost of the computation can be reduced.

3.3 Level Properties

In addition to capabilities, every coordinate hierarchy level may possess one or more of five *properties*: *full*, *ordered*, *unique*, *branchless*, and *compact*. These describe various characteristics of levels such as whether coordinates are arranged in order or whether each coordinate has no duplicate; we explain each property in detail below. Properties of a coordinate hierarchy level reflect invariants that are explicitly enforced or implicitly assumed by the underlying physical index. A level that corresponds to the column dimen-

sion of a CSR matrix, for instance, is both ordered and unique (Figure 3-5), which reflects how the standard CSR format stores every non-zero coordinate just once and stores the components in memory in increasing order of the coordinate values.

Our code generation technique relies on these properties to emit code that is optimized for its inputs. For example, knowing that an input tensor format encodes only unique coordinates enables our algorithm to avoid emitting code to aggregate duplicate values, resulting in more efficient code.

Full A level is full if every collection of coordinates that share the same ancestor coordinates contains all possible coordinates along the dimension corresponding to that level. For instance, a level that represents a dense array vector (Figure 3-1a) necessarily encodes every coordinate and is thus full. On the other hand, a level that represents a sparse vector (Figure 3-1b) is not full as sparse vectors only store non-zero coordinates.

Unique A level is unique if every collection of coordinates that share the same ancestor coordinates contains no duplicate coordinates. For instance, a level that represents the row dimension of a CSR matrix (Figure 3-2b) necessarily encodes every coordinate just once and is thus unique. By contrast, a level that represents the row dimension of a COO matrix (Figure 3-2a) can store the same coordinate more than once and is thus not unique.

Ordered A level is ordered if coordinates sharing the same ancestor coordinates are arranged in increasing value, coordinates with different ancestor coordinates are ordered lexicographically by their ancestor coordinates, and duplicates are ordered by their parents' positions. For example, a level that represents a standard sparse vector stores coordinates in increasing order and is thus ordered. By contrast, a level that represents a hash map vector (Figure 3-1c) is not ordered since the underlying hash table stores coordinates in hash order, which does not necessarily match the order of the coordinate values.

Branchless A level is branchless if no coordinate has a sibling and every coordinate in the previous level has a child. For example, the coordinate hierarchy for a COO matrix consists strictly of chains of coordinates, making the lower level branchless. On the other

Table 3.2: Properties of each of the six level types listed in Section 3.1. ✓ indicates that a particular level type always possesses a particular property, while (✓) indicates that a level type can be configured to either possess or not possess a particular property.

Level Type	Full	Unique	Ordered	Branchless	Compact
Dense	✓	(✓)	(✓)		✓
Range		(✓)	(✓)		
Compressed	(✓)	(✓)	(✓)		✓
Singleton	(✓)	(✓)	(✓)	✓	✓
Offset		(✓)	(✓)	✓	
Hashed		(✓)			

hand, a level that represents the column dimension of a CSR matrix can have multiple coordinates that share the same parent and is thus not branchless.

Compact A level is compact if no two coordinates are separated by a position that does not encode a coordinate (i.e., a node with no number). For instance, a level that represents a sparse vector encodes coordinates in a single contiguous range of positions and is thus compact. By contrast, a level representing a hash map vector can have unlabeled positions that correspond to empty hash table buckets and is thus not compact.

Table 3.2 describes properties for each of the six level types listed in Section 3.1. In certain cases, denoted by (✓), a coordinate hierarchy level may be optionally configured to possess a particular property based on the application. Configurable properties reflect invariants that are not tied to how a physical index encodes coordinates; for instance, the `idx` array in compressed levels typically assume coordinates are stored in order when used in common tensor formats like CSR, but the same data structure can just as easily store coordinates out of order. By contrast, non-configurable properties correspond to invariants that arise directly from how a physical index encodes coordinates; for example, the hash table that underlies hashed levels has to store coordinates in hash order in order to provide constant-time random access, thereby making it impossible for a hashed level to be ordered in practical use cases. Figure 3-5 shows exactly how level types with configurable properties can be configured to represent concrete tensor formats.

3.4 Output Assembly

The capabilities described in Section 3.2 all relate to accessing coordinate hierarchies that have already been constructed. A level in a coordinate hierarchy may also provide the append capability or the insert capability, which enable new coordinates and edges to be added to the level. These capabilities provide an abstraction for assembling, in a format-agnostic manner, physical indices that store the result of a computation.

Insert Capability The insert capability permits coordinates to be inserted anywhere into a coordinate hierarchy level and is defined in terms of four level functions:

```
insert_coord(p_k, i_1, ..., i_k) -> void

init_insert(p_{k-1}_max, p_k_max) -> void
finalize_insert(p_{k-1}_max, p_k_max) -> void
nonzeros(p_{k-1}_max) -> p_k_max
```

New coordinates can be inserted into an output level by calling `insert_coord`, which intuitively adds a coordinate at position p_k that encodes the subtensor $A(i_1, ..., i_k, :, ..., :)$, with positions given by the `locate` function as inputs. This requires the output level to provide the locate capability but allows result coordinates to be inserted in any order.

`init_insert` defines how the data structures that encode an output level should be initialized before starting an actual computation, while `finalize_insert` defines any post-processing needed after the result has been fully computed in order to finish assembling the aforementioned data structures. `nonzeros` returns an upper bound on the positions of coordinates in an output level. All three functions take, as each input argument $\mathtt{p}_i\mathtt{\_max}$, the value returned by `nonzeros` for the i-th level if it is assembled with the insert capability or the number of coordinates that have been appended to the i-th level if it is assembled with the append capability, which we describe next.

Append Capability The append capability permits coordinates to be appended to the end of a coordinate hierarchy level and is also defined in terms of four level functions:

```
append_coord(p_k, i_1, ..., i_k) -> void
append_edges(p_{k-1}, p_k_begin, p_k_end) -> void

init_append(p_{k-1}_max, p_k_max) -> void
finalize_append(p_{k-1}_max, p_k_max) -> void
```

New coordinates can be appended to an output level by calling `append_coord`, which like `insert_coord` adds a coordinate at position p_k that encodes the subtensor $A(i_1, ..., i_k, :, ..., :)$, with successively incremented values of p_k as inputs. Newly appended coordinates can then be attached to the rest of the coordinate hierarchy by calling `append_edges`, which inserts edges that connect all coordinates positioned between p_k`_begin` (inclusive) and p_k`_end` (exclusive) to the coordinate at position p_{k-1} in the previous level. Unlike with the insert capability though, making use of the append capability requires result coordinates to be enumerated and appended in order.

`init_append` and `finalize_append` serve identical purposes as `init_insert` and `finalize_insert` respectively and take the same input arguments.

Table 3.3 lists the assembly capabilities provided by four level types that support assembly along with valid definitions of the corresponding level functions. Following the semantics described above, we can, for example, assemble a CSR matrix A from another DCSR matrix B by calling the appropriate level functions as demonstrated below:

```
1  pA2 = 0;
2  A1_init_insert(1, A1_nonzeros(1));
3  A2_init_append(A1_nonzeros(1), pA2);
4  for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; ++pB1) {
5    int i = B1_idx[pB1];
6    int pA1 = i;
7    int pA2_begin = pA2;
8    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; ++pB2) {
9      int j = B2_idx[pB2];
10     A2_append_coord(pA2++, i, j);
11   }
12   A2_append_edges(pA1, pA2_begin, pA2);
13   A1_insert_coord(pA1, i);
```

Table 3.3: Assembly capabilities provided by four of the level types listed in Section 3.1 that support assembly, and definitions of their corresponding level functions.

Level Type	Supported Capabilities	Level Function Definitions

Dense	Insert	`insert_coord(p_k, i_1, ..., i_k):`
		`  // do nothing`
		
		`init_insert(p_{k-1}_max, p_k_max):`
		`  // do nothing`
		
		`finalize_insert(p_{k-1}_max, p_k_max):`
		`  // do nothing`
		
		`nonzeros(p_{k-1}_max):`
		`  return p_{k-1}_max * N_k`

Compressed	Append	`append_coord(p_k, i_1, ..., i_k):`
		`  idx[p_k] = i_k`
		
		`append_edges(p_{k-1}, p_k_begin, p_k_end):`
		`  pos[p_{k-1} + 1] = p_k_end - p_k_begin`
		
		`init_append(p_{k-1}_max, p_k_max):`
		`  for (int p_{k-1} = 0; p_{k-1} <= p_{k-1}_max; ++p_{k-1}) {`
		`  pos[p_{k-1}] = 0`
		`  }`
		
		`finalize_append(p_{k-1}_max, p_k_max):`
		`  int cumsum = pos[0]`
		`  for (int p_{k-1} = 1; p_{k-1} <= p_{k-1}_max; ++p_{k-1}) {`
		`  cumsum += pos[p_{k-1}]`
		`  pos[p_{k-1}] = cumsum`
		`  }`

Singleton	Append	`append_coord(p_k, i_1, ..., i_k):`
		`  idx[p_k] = i_k`
		
		`append_edges(p_{k-1}, p_k_begin, p_k_end):`
		`  // do nothing`
		
		`init_append(p_{k-1}_max, p_k_max):`
		`  // do nothing`
		
		`finalize_append(p_{k-1}_max, p_k_max):`
		`  // do nothing`

Hashed	Insert	`insert_coord(p_k, i_1, ..., i_k):`
		`  idx[p_k] = i_k`
		
		`init_insert(p_{k-1}_max, p_k_max):`
		`  for (int p_k = 0; p_k < p_k_max; ++p_k) {`
		`  idx[p_k] = -1`
		`  }`
		
		`finalize_insert(p_{k-1}_max, p_k_max):`
		`  // do nothing`
		
		`nonzeros(p_{k-1}_max):`
		`  return p_{k-1}_max * W_k`

```
14 }
15 A2_finalize_append(A1_nonzeros(1), pA2);
16 A1_finalize_insert(1, A1_nonzeros(1));
```

where level functions prefixed with `A1_` and `A2_` refer to the versions defined for dense and compressed levels respectively (since the CSR matrix hierarchy consists of a dense level and a compressed level). In this example, the calls to `A2_append_coord` assemble the output `idx` array while the calls to `A2_init_append`, `A2_append_edges`, and `A2_finalize_append` assemble the output `pos` array as required.

Some level types permit multiple ways of implementing an assembly capability. Below, for instance, is another way to implement the append capability for a compressed level, which is also valid as long as the preceding level in the hierarchy also has append capability:

```
append_coord(p_k, i_1, ..., i_k):
  idx[p_k] = i_k

append_edges(p_{k-1}, p_k_begin, p_k_end):
  pos[p_{k-1} + 1] = p_k_end

init_append(p_{k-1}_max, p_k_max):
  pos[0] = 0

finalize_append(p_{k-1}_max, p_k_max):
  // do nothing
```

This alternative definition is more restrictive than the one presented in Table 3.3—it cannot be used to assemble a CSR matrix, as an example—but can provide better performance as it fully assembles the `pos` array in one shot and does not require any post-processing pass. Again, by hiding these implementation choices behind an abstract interface, we make it possible to develop a generic code generation algorithm that naturally supports all these alternatives by simply emitting code like the one shown previously.

Chapter 4

Code Generation

In this chapter, we describe a code generation technique that emits efficient code to compute any compound tensor algebra expression, where the operands can be stored in any combination of tensor formats assembled from level formats described in Chapter 3. These include, but are not limited to, the tensor storage formats we examined in Chapter 2. Our algorithm extends the code generation technique Kjolstad et al. proposed in [34] to emit code that efficiently iterates and merges coordinate hierarchies, and which can then be specialized to efficiently compute with specific tensor formats without requiring the code generator to directly reason about those formats. This separation between tensor formats and the code generation algorithm, which the coordinate hierarchy abstraction enforces, limits the complexity of the algorithm and ensures it can be reasonably maintained and potentially extended to support even more formats.

4.1 Iteration Graphs and Merge Lattices

Our code generation technique takes as input a tensor algebra expression in a variant of the tensor index notation developed by Ricci-Curbastro and Levi-Civita [53], which describes how each component in the output of a tensor computation can be computed in terms of components in the operands. For example, matrix-vector multiplication can be

expressed in tensor index notation as

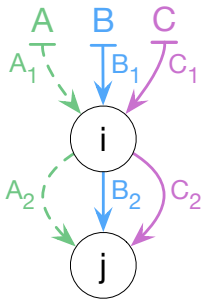
$$y_i = \sum_j A_{ij} x_j,$$

which makes explicit that every i-th row in the result is computed as the inner product of the corresponding row in the input matrix A with input vector j. Similarly, the addition of two matrices can be expressed as

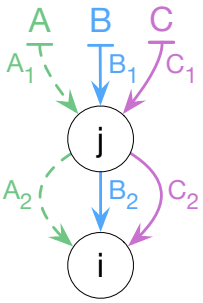
$$A_{ij} = B_{ij} + C_{ij}.$$

Computing a tensor algebra expression in this form requires iterating over the merged iteration space of the operands dimension by dimension. For instance, efficient code that adds two sparse matrices must logically iterate only rows that have non-zeros in either matrix and, for each row, iterate only the columns that are non-zero in either matrix. How exactly tensor dimensions should be merged depends on the computation. Multiplication requires iterating over the intersection of the operands as the result is non-zero only if both operands are non-zero. Addition, by contrast, requires iterating over the union of the operands as the result is non-zero if at least one of the operands is non-zero.

The proper order in which to iterate over dimensions in the merged iteration space can be determined with the aid of an *iteration graph*, which is an intermediate representation Kjolstad et al. proposed that describes how to iterate over non-zero inputs of a tensor algebra expression [34]. More formally, the iteration graph for a tensor algebra expression consists of a set of index variables that appear in the expression and a set of directed tensor paths that represent accesses into input and output tensors. Each tensor path connects index variables that are used in the corresponding tensor access and is ordered based on the order of index variables in the access expression as well as the order in which dimensions of the accessed tensor are stored. Determining the order in which dimensions should be merged to efficiently compute an expression then reduces to ordering index variables into a forest such that every tensor path edge goes from an index variable that is higher up to



(a) Iteration graph for CSR matrix addition



(b) Iteration graph for CSC matrix addition

Figure 4-1: Iteration graphs for matrix addition $A_{ij} = B_{ij} + C_{ij}$, where the input and output matrices are (a) all stored in the CSR format or (b) all stored in the CSC format.

an index variable that is lower down. Such a forest ordering, by construction, must satisfy all constraints on efficiently accessing tensor operands that are imposed by their formats.

Figure 4-1 shows examples of iteration graphs, arranged in their forest ordering, for the addition of two CSR matrices and the addition of two CSC matrices. As we saw in Figure 3-2b, a CSR matrix encodes its row dimension before its column dimension in the corresponding coordinate hierarchy (which reflects how row coordinates are used to index into the pos array), so all tensor paths in Figure 4-1a have edges that point from index variable i to j. Thus, from the iteration graph's forest ordering, we can determine that efficiently computing CSR matrix addition requires iterating the row dimensions before the column dimensions. By contrast, as we saw in Figure 3-2c, a CSC matrix encodes its column dimension before its row dimension (which reflects how column coordinates are used to index into the pos array instead), so all tensor paths in Figure 4-1b have edges that point from j to i. Thus, in this case, we can determine from the iteration graph's forest ordering that efficiently computing CSC matrix addition requires iterating the column dimensions before the row dimensions.

For each dimension in the merged iteration space, its corresponding *merge lattice*–another intermediate representation proposed by Kjolstad et al.—describes what loops are needed to compute an arbitrary combination of intersection and union merges involving the input tensor dimensions. Each point in the ordered lattice encodes a set of tensor dimensions containing non-zero coordinates that needs to be merged with one loop, while lattice points that are dominated by the given lattice point encode different cases that

(a) Merge lattice for i

(b) Merge lattice for j

Figure 4-2: Merge lattices for CSR matrix addition $A_{ij} = B_{ij} + C_{ij}$.

the aforementioned loop must consider when computing the merge. Every path from the top lattice point to the bottom lattice point represents a sequence of loops that might have to be executed at runtime in order to fully merge the input dimensions. Kjolstad et al. [34] gives a bottom-up recursive algorithm that constructs a merge lattice for any index expression and any given index variable.

Figure 4-2 shows examples of merge lattices that describe the addition of two CSR matrices. The merge lattice describing how to merge the column dimensions, shown in Figure 4-2b, contains three lattice points that yield three separate loops. The first loop co-iterates over the column dimensions of input matrices B and C and merges the two while they both contain unprocessed non-zeros, while the remaining two loops iterate over remaining values of B or C after at least one input has been exhausted. The merge lattice for the row dimensions, shown in Figure 4-2a, contains just one lattice point as both input dimensions are dense and thus must share the same iteration space.

4.2 Merge Lattice Optimizations

Kjolstad et al. proposed a set of optimizations that can be applied to merge lattices in order to obtain simplified lattices like the one shown in Figure 4-2a for matrix addition and thus yield more optimized code [34]. All of those optimizations were formulated with respect to dimensions stored in what we classify as dense and compressed level formats, though it is straightforward to generalize those optimizations to other level types that our technique supports. In particular, we can optimize merging of any number of dimensions encoded as

full coordinate hierarchy levels (as opposed to only dense dimensions) by iterating over just one of the full dimensions, as long as the others all support the locate capability. We can also safely remove all lattice points below the top point that do not merge every full dimension, as full dimensions are supersets of any sparse iteration space.

4.3 Merging Coordinate Hierarchy Levels

Merging tensor dimensions is equivalent to merging the corresponding coordinate hierarchy levels that represent those dimensions. The most efficient method for merging coordinate hierarchy levels depends on the properties of the levels and the capabilities that the levels support. Consider, for instance, the component-wise multiplication of two vectors x and y, which requires iterating over the intersection of the two coordinate hierarchy levels that encode their non-zero coordinates. Figure 4-3 shows optimal strategies for computing the intersection merge depending on whether the input levels are ordered or unique and whether they support the locate capability; these are the same strategies our code generation algorithm selects. The rest of this section focuses on standard versions of two high-level techniques, which require input levels with specific properties or capabilities and are indicated by green and red/blue squares without circles or diamonds in Figure 4-3. Section 4.4 examines how these high-level techniques can be applied to merge any coordinate hierarchy levels regardless of their properties and capabilities.

If both inputs support iterating unique coordinates in order but do not support the locate capability, we can co-iterate over the two inputs (lines 3–14) and compute a new output component whenever we encounter non-zero components in both inputs that share the same coordinate (7–11):

```
1  int px = x_pos[0];
2  int py = y_pos[0];
3  while (px < x_pos[1] && py < y_pos[1]) {
4    int ix = x_idx[px];
5    int iy = y_idx[px];
6    int i = min(ix, iy);
7    if (ix == i && iy == i) {
```

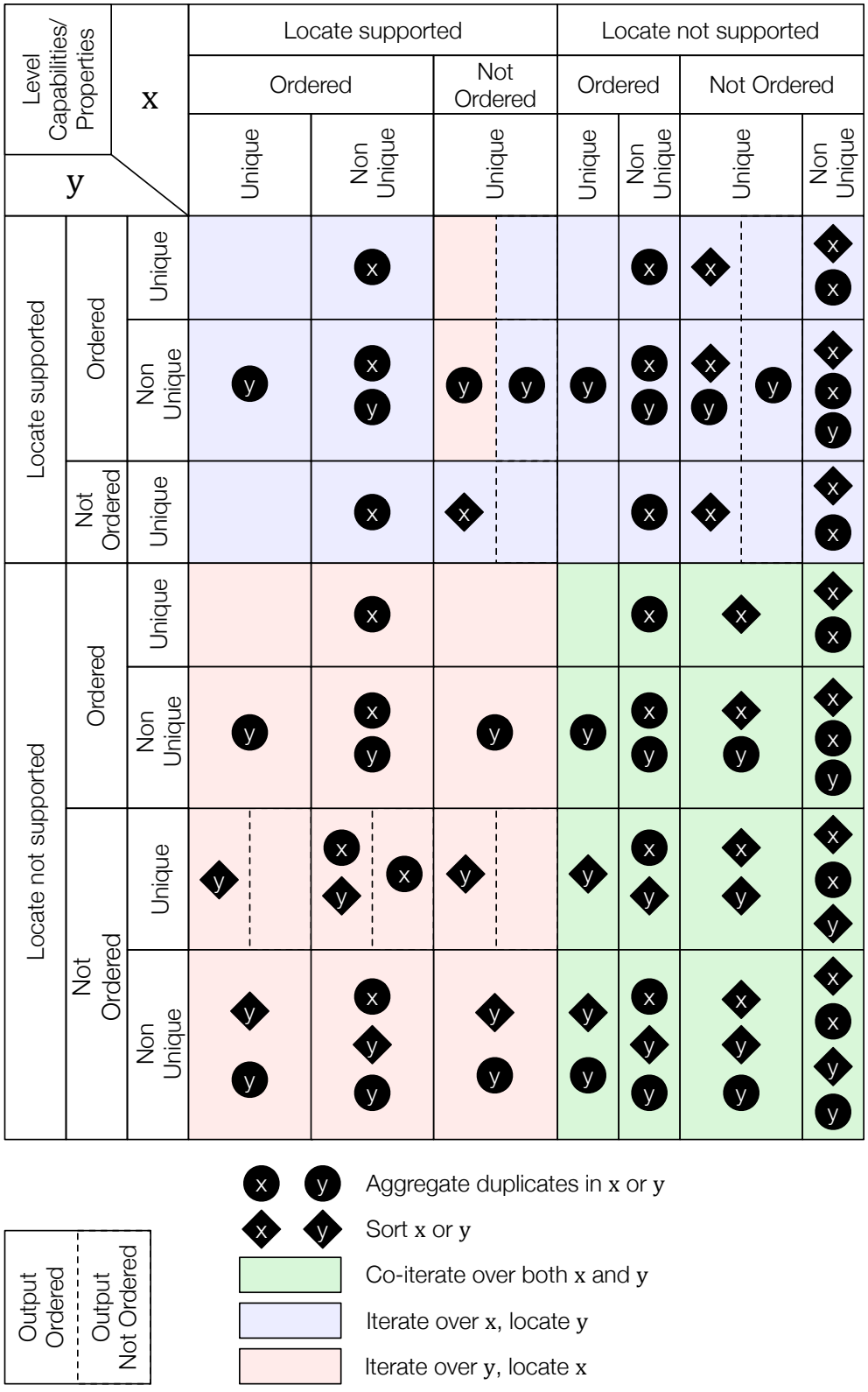


Figure 4-3: The optimal strategies for computing the intersection merge of two vectors x and y depending on whether they provide locate capability and whether they are ordered and unique. y is assumed to not be a strict subset of x.

```
 8     double x_val = x_vals[px];
 9     double y_val = y_vals[py];
10     z_vals[i] = x_val * y_val;
11   }
12   if (ix == i) ++px;
13   if (iy == i) ++py;
14 }
```

This technique generalizes the two-way merge algorithm used in merge sort [36, Chapter 5.2.4] but still requires both inputs to be ordered and unique.

However, if one of the inputs, y, supports the locate capability (e.g., if it is dense), then we can instead iterate the non-zero coordinates in x (lines 1–2) and, for each coordinate, locate the corresponding coordinate in y (line 4):

```
1 for (int px = x_pos[0]; px < x_pos[1]; ++px) {
2   int i = x_idx[px];
3   double x_val = x_vals[px];
4   double y_val = y_vals[i];
5   z_vals[i] = x_val * y_val;
6 }
```

This alternative technique (iterate-and-locate) reduces the complexity of the merge from $O(\text{nnz}(x) + \text{nnz}(y))$ to $O(\text{nnz}(x))$, which is a significant improvement if y has many more non-zeros than x (e.g., if y is dense). Moreover, this technique does not require the coordinates in y to be stored in order. In fact, we do not even need to enumerate the coordinates in x in order, as long as there are no duplicates and we do not need to enumerate output components in order (e.g., if the output supports the insert capability). This alternative technique is thus ideal for merging unordered dimensions.

We can generalize and combine the two techniques described above to, with the help of merge lattices, compute arbitrarily complex merges involving unions and intersections of any number of tensor dimensions. If a merge lattice contains multiple lattice points, then each lattice point can be converted to a loop that co-iterates over all the corresponding hierarchy levels that need to be merged. However, if a merge lattice contains just one lattice point (excluding the bottom), then it can be converted to a loop that co-iterates over

only the hierarchy levels that need to be merged but do not support the locate capability, and which locates into the other levels that also need to be merged.

4.4 Iterator Conversion

As we saw in the previous section, efficient algorithms to merge coordinate hierarchy levels exist when they are all ordered and unique (i.e., co-iteration), or when the merge operation is an intersection and all of the unordered levels support the locate capability (i.e., iterate-and-locate). Iterator conversion describes a set of transformations that convert iterators over unordered or non-unique hierarchy levels without the locate capability to new iterators with the required properties on the fly, so that the aforementioned algorithms can be used where they otherwise could not. Here we describe two types of iterator conversion, which can be arbitrarily composed as needed and which are sufficient to support merging any hierarchy levels using the two high-level techniques described in Section 4.3.

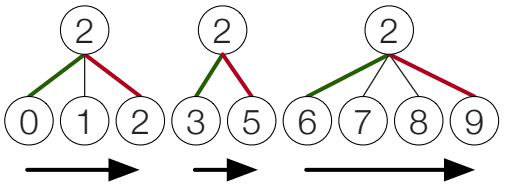
Deduplication The existence of duplicate coordinates poses a complication for merging as it results in repeated visits of the same points in the iteration space. Deduplication, an example of which is shown below, removes duplicates from an iterator over an ordered but non-unique coordinate hierarchy level using a loop that scans ahead in the level (lines 6–9) and aggregates duplicate coordinates (line 7).

```
1  int p = pos[0];
2  while (p < pos[1]) {
3    int i = idx[p];
4    double v = vals[p];
5    int p_segend = p + 1;
6    while (p_segend < pos[1] && idx[p_segend] == i) {
7      v += vals[p_segend];
8      ++p_segend;
9    }
10   printf("x(%d) = %f\n", i, v);
11   p = p_segend;
12 }
```

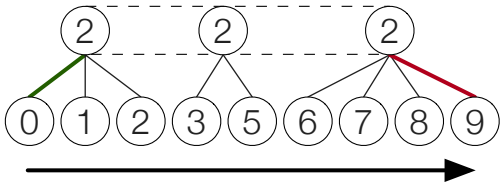
Thus, for the purpose of merging, the emitted code can treat the resultant iterator as one that does not enumerate duplicates. Figure 4-3 identifies, with black circles, the cases where deduplication needs to be applied to one or both operands.

If the non-unique level is the bottom-most level in a coordinate hierarchy, as is the case in the previous example, then values at the duplicate coordinates are aggregated using a summation reduction. Otherwise, the emitted code needs to aggregate all children of the duplicate coordinates in such a way that only a single iterator is needed to iterate over them. In the general case, as shown in the example below, this requires assembling a scratch array list that stores all the child coordinates (lines 7–16) and that can be iterated with a single iterator (lines 18–23).

```
1  typedef struct { int i; int p; } coord;
2  coord coords[];
3
4  int p1 = pos1[0];
5  while (p1 < pos1[1]) {
6    int i = idx1[p];
7    int nj = 0;
8    int p1_segend = p1 + 1;
9    while (p1_segend < pos1[1] && idx1[p1_segend] == i) {
10     for (int p2 = pos2[p1]; p2 < pos2[p1 + 1]; ++p2, ++nj) {
11       coords[nj].p = p2;
12       coords[nj].i = idx2[p2];
13     }
14     ++p1_segend;
15   }
16   sort(coords, nj);
17
18   for (int n = 0; n < nj; ++n) {
19     int p2 = coords[n].p;
20     int j = coords[n].i;
21     double v = vals[p2];
22     printf("A(%d, %d) = %f\n", i, j, v);
23   }
24   p1 = p_segend;
25 }
```



(a) Each coordinate position iterator iterates over the children of a duplicate coordinate.



(b) Chained iterator iterates over all children of all duplicate coordinates.

Figure 4-4: Iterator chaining chains together coordinate position iterators over distinct collections of children of duplicate coordinates (a) into a single iterator over all the children (b) without needing a scratch array. Each arrow represents an iterator with its starting and ending bounds indicated by green and red edges respectively.

However, if the child coordinates are stored in a level that supports coordinate position iteration and if both that level and the non-unique level before it are ordered and compact, then instead the emitted code can chain together iterators that each iterate over the children of one duplicate coordinate (Figure 4-4a) into a single iterator that iterates over all children of all the duplicate coordinates (Figure 4-4b). The starting bound of the resultant iterator would simply be the starting bound of the iterator over the first collection of children, while the ending bound of the resultant iterator would be the ending bound of the iterator over the last collection of children. The resultant chained iterator provides the same interface as regular coordinate position iterators and can thus participate in merging in the same way as any other iterator without needing to assemble a scratch array first. The code below demonstrates this more concretely; p2 represents the chained iterator over all child coordinates of i, with its starting and ending bounds computed from the first position that encodes i (p1) and the last position that encodes i (p1_segend) respectively (line 12).

```
1  struct { int i; int p; } coord;
2  coord coords[];
3
4  int p1 = pos1[0];
5  while (p1 < pos1[1]) {
6    int i = idx1[p];
7    int p1_segend = p1 + 1;
8    while (p1_segend < pos1[1] && idx1[p1_segend] == i) {
9      ++p1_segend;
10   }
```

```
11
12   for (int p2 = pos2[p1]; p2 < pos2[p1_segend]; ++p2) {
13     int j = idx2[p2];
14     double v = vals[p2];
15     printf("A(%d, %d) = %f\n", i, j, v);
16   }
17   p1 = p_segend;
18 }
```

Reordering An essential precondition for co-iterating hierarchy levels is that the input coordinates can be enumerated in order. Reordering, an example of which is shown below, assembles a scratch array list that stores an ordered copy of an unordered coordinate hierarchy level (lines 4–9) and replaces the iterator over the unordered level with an iterator over the ordered copy (lines 11–13).

```
1  typedef struct { int i; int p; } coord;
2  coord coords[];
3
4  int nnz = 0;
5  for (int p = pos[0]; p < pos[1]; ++p, ++nnz) {
6    coords[nnz].p = p;
7    coords[nnz].i = idx[p];
8  }
9  sort(coords, nnz);
10
11 for (int n = 0; n < nnz; ++n) {
12   int p = coords[n].p;
13   int i = coords[n].i;
14   double v = vals[p];
15   printf("x(%d) = %f\n", i, v);
16 }
```

Thus, the code generation algorithm can emit code that merges an unordered level by co-iterating over the ordered copy. Figure 4-3 identifies, with black diamonds, the cases where reordering needs to be applied to one or both operands.

4.5 Code Generation Algorithm

Figure 4-5 shows our code generation algorithm, which incorporates all of the concepts we presented in the previous sections. Each part of the algorithm is labeled from 1 to 11; throughout the discussion of the algorithm in the rest of this section, we will identify relevant parts using these labels.

The algorithm emits code that iterates over the proper intersections and unions of the input tensors by calling relevant access capability level functions, computes values at points in the merged iteration space, and assembles an output tensor by calling the relevant assembly capability level functions. The generated code is then specialized to work with specific tensor formats by inlining all the level function calls. This approach limits the complexity of the code generation mechanism, as it only needs to reason about iterating and merging coordinate hierarchy levels with a finite set of distinct capabilities as opposed to unbounded combinations of specific types of physical indices. The result is an algorithm that supports disparate tensor formats and that does not need modification to add support for more level types and tensor formats.

The algorithm recursively calls itself on index variables in the tensor algebra expression to be computed, in the order given by the corresponding iteration graph. At each recursion level, the algorithm generates code for one index variable iv in the input index expression. The algorithm begins by emitting code that calls the appropriate coordinate value iteration or coordinate position iteration level functions to initialize iterators into coordinate hierarchy levels representing input tensor dimensions, and that performs any necessary iterator conversion described in Section 4.4 (1, 2).

The algorithm also constructs a merge lattice at every recursion level for the corresponding input expression and index variable iv. For every point L_p in this merge lattice, the algorithm emits a loop that merges coordinate hierarchy levels representing input tensor dimensions associated with the lattice point (3). Within each loop, the generated code dereferences iterators over the coordinate hierarchy levels to be merged (4, 6, 9), making sure to not process coordinates that are not actually encoded in the levels (5). Each merged coordinate is then computed (7) and, if the merge is an intersection, used to index

into levels that can be accessed with the locate capability (8). At the very end, the algorithm emits appropriate code to advance the iterators that referenced merged coordinates (11).

The algorithm also generates code to compute output values and to assemble output indices (10). The latter involves emitting code to keep track of where to insert or append new result coordinates in the output, as well as emitting code that calls the appropriate assembly level functions to assemble physical indices for the output (`emit-index-assembly`). The algorithm emits specialized compute and assembly code for each point in the merge lattice that is dominated by the loop lattice point L_p, which handles the case where the corresponding subset of input dimensions contains non-zeros at the same coordinate.

Figure 4-6 and Figure 4-7 shows examples of code that our technique generates for adding a CSR matrix to a sorted COO matrix with no empty row. The algorithm in Figure 4-5 emits exactly what is shown in Figure 4-6 with only the calls to level functions, which frees the algorithm from needing to reason about merging physical indices like the `idx` arrays in the input matrices. Work that is required to specialize the emitted code to actually merge physical indices associated with the CSR and COO formats (as shown in Figure 4-7) is offloaded entirely to the inlining pass, which can do this completely mechanically. More examples of code that our technique generates are shown in Appendix A.

Fusing Iterators By default, at every recursion level, the algorithm emits loops that iterate over a single coordinate hierarchy level of each input tensor. However, an optimization that improves performance when computing with formats such as COO involves emitting code that simultaneously iterates over multiple coordinate hierarchy levels. The algorithm implements this optimization by fusing iterators over branchless levels with iterators over their preceding levels, as long as the associated merges are intersections and other merged levels can be accessed with the locate capability. The algorithm thus avoids emitting loops for levels that are accessed by fused iterators (3), which eliminates unnecessary branching costs. For some computations, this optimization can transform the generated kernel from a gather code that enumerates each output non-zero once to a scatter code that accumulates into non-zero outputs, in which case the algorithm ensures that the output coordinate hierarchy levels can be accessed with the locate capability.

```
code-gen(index-expr, iv):
  let L = merge-lattice(index-expr, iv)

  for Dj in coord-value-iteration-dims(L):
    emit   "int ivDj, int Dj_end = coord_iter_Dj(ivD1,...,ivDj-1);"
  for Dj in coord-position-iteration-dims(L):
    if Dj-1 is unique:
1     emit "int pDj, int Dj_end = pos_iter_Dj(pDj-1);"
    else:
      emit "int pDj, _ = pos_iter_Dj(pDj-1);"
      emit "_, int Dj_end = pos_iter_Dj(Dj-1_segend - 1);"

  for Dj in noncanonical-dims(L):
    emit-scratch-array-assembly(Dj)
2   if Dj is unordered:
      emit "sort(Dj_scratch, 0, Dj_end);"
    emit   "int itDj = 0;"

  for Lp in L:
    if iterator for each Dj in merged-dims(Lp) is unfused:
3     let mdims = merged-dims(Lp)
      emit “while(all(["{p|it|iv}Dj < Dj_end" for Dj in mdims])) {”

    for Dj in coord-value-iteration-dims(Lp):
      emit   “int pDj, bool fDj = coord_access_Dj(pDj-1,...,ivDj);”
      emit   "while (!fDj && ivDj < Dj_end)"
      emit     "pDj, fDj = coord_access_Dj(pDj-1,...,++ivDj);"
4   for Dj in coord-position-iteration-dims(Lp):
      emit   “int ivDj, bool fDj = pos_access_Dj(pDj,...,ivDj-1);”
      emit   "while (!fDj && pDj < Dj_end)"
      emit     "ivDj, fDj = pos_access_Dj(++pDj,...,ivDj-1);"
    let cmdims = canonical-merged-dims(Lp)
5   emit     "if(all([“fDj” for Dj in cmdims])) {"
    for Dj in noncanonical-dims(Lp):
6     emit     "int ivDj = Dj_scratch[itDj].i;"
      emit     "int pDj = Dj_scratch[itDj].p;"

7   emit       “int iv = min([“ivDj” for Dj in merged-dims(Lp)]);”
    for Dj in locate-supported-dims(Lp):
8     emit     "int pDj, bool fDj = locate_Dj(pDj-1,...,iv);"
```

```
for Dj in noncanonical-dims(Lp) U coord-position-iteration-dims(Lp):
  emit     "int Dj_segend = pDj + 1;"
  if Dj is not unique and iterator for Dj is unfused:
    emit-deduplication-loop(Dj)

emit-available-expressions(index-expr, iv)
if result dimension Dj supports insert and iv indexes Dj:
  emit     "int pDj, _ = locate_Dj(pDj-1,...,iv);"
for Lq in sub-lattice(Lp):  # a case per lattice point below Lp
  let mdims = merged-dims(Lq) \ full-dims(Lq)
  let locdims = locate-supported-dims(Lq) \ full-dims(Lq)
  emit     “if (all([“ivDj == iv” for Dj in mdims]) &&
               all(["fDj" for Dj in locdims])) {”
  for child-iv in children-in-iteraton-graph(iv):
    code-gen(expression(Lq), child-iv)
  emit-reduction-compute()
  emit-index-assembly()
  emit-compute()
  if result dimension Dj supports append and iv indexes Dj:
    emit     “pDj++;”
  emit     “}”

for Dj in merged-dims(Lp):
  if Dj is not full:
    emit   "if (ivDj == iv) "
  if Dj in coord-value-iteration-dims(Lp):
    emit     "ivDj++;"
  else:
    emit     "{p|it}Dj = Dj_segend;"
emit     "}"
if iterator for each Dj in merged-dims(Lp) is unfused:
  emit "}"
```

Figure 4-5: Recursive algorithm for generating code that computes tensor algebra expressions on coordinate hierarchies. `coordinate-value-iteration-dims` and `coordinate-position-iteration-dims` all exclude dimensions in `noncanonical-dims`. All three and `merged-dims` also exclude dimensions in `locate-supported-dims` for intersection merges. In this context, canonical dimensions refer to those that do not need to be backed up by a scratch array list as described in Section 4.4.

```
int iB1, int B1_end = B1_coord_iter();
int pC1, int C1_end = C1_pos_iter(0);
while (iB1 < B1_end && pC1 < C1_end) {
  int pB1 = B1_coord_access(0, iB1);
  int iC1 = C1_pos_access(pC1);
  int i = iB1;
  int C1_segend = C1_pos + 1;
  while (C1_segend < C1_end && C1_pos_access(C1_segend) == <i, true>)
    C1_segend++;
  int pA1, _ = A1_locate(0, i);
  int pB2, int B2_end = B2_pos_iter(pB1);
  int pC2, _ = C2_pos_iter(pC1);
  _, int C2_end = C2_pos_iter(C2_segend - 1);
  while (pB2 < B2_end && pC2 < C2_end) {
    int jB2 = B2_pos_access(pB2, i);
    int jC2 = C2_pos_access(pC2, i);
    int j = min(jB2, jC2);
    int B2_segend = pB2 + 1;
    int C2_segend = pC2 + 1;
    int pA2 = A2_locate(pA1, i, j);
    if (jB2 == j && jC2 == j) {
      A_vals[pA2] = B_vals[pB2] + C_vals[pC2];
    } else if (jB == j) {
      A_vals[pA2] = B_vals[pB2];
    } else if (jC == j) {
      A_vals[pA2] = C_vals[pC2];
    }
    if (jB2 == j) pB2 = B2_segend;
    if (jC2 == j) pC2 = C2_segend;
  }
  while (pB2 < B2_end) {
    int jB2 = B2_pos_access(pB2, i);
    int j = jB2;
    int B2_segend = pB2 + 1;
    int pA2 = A2_locate(pA1, i, j);
    A_vals[pA2] = B_vals[pB2];
    pB2 = B2_segend;
  }
  while (pC2 < C2_end) {
    int jC2 = C2_pos_access(pC2, i);
    int j = jC2;
    int C2_segend = pC2 + 1;
    int pA2 = A2_locate(pA1, i, j);
    A_vals[pA2] = C_vals[pC2];
    pC2 = C2_segend;
  }
  iB1++;
  pC1 = C1_segend;
}
```

Figure 4-6: Sparse matrix addition kernel, which adds a CSR matrix to a sorted COO matrix with no empty row, emitted by our code generation algorithm. The labels on the margin map each line in the generated code to the part of the algorithm that emits it. For clarity, we simplified the generated code by eliminating trivial if, while, and min statements. Tuples are denoted with angle brackets (e.g., `<i, true>`).

```
int iB1 = 0;
int B1_end = B1_size;
int pC1 = C1_pos[0];
int C1_end = C1_pos[1];
while (iB1 < B1_end && pC1 < C1_end) {
  int pB1 = (0 * B1_size) + iB1;
  int iC1 = C1_idx[pC1];
  int i = iB1;
  int C1_segend = C1_pos + 1;
  while (C1_segend < C1_end && C1_idx[C1_segend] == i)
    C1_segend++;
  int pA1 = (0 * A1_size) + i;
  int pB2 = B2_pos[pB1];
  int B2_end = B2_pos[pB1 + 1];
  int pC2 = pC1;
  int C2_end = C2_segend;
  while (pB2 < B2_end && pC2 < C2_end) {
    int jB2 = B2_idx[pB2];
    int jC2 = C2_idx[pC2];
    int j = min(jB2, jC2);
    int B2_segend = pB2 + 1;
    int C2_segend = pC2 + 1;
    int pA2 = (pA1 * A2_size) + j;
    if (jB2 == j && jC2 == j) {
      A_vals[pA2] = B_vals[pB2] + C_vals[pC2];
    } else if (jB == j) {
      A_vals[pA2] = B_vals[B2_pos];
    } else if (jC == j) {
      A_vals[pA2] = C_vals[C2_pos];
    }
    if (jB2 == j) pB2 = B2_segend;
    if (jC2 == j) pC2 = C2_segend;
  }
  while (pB2 < B2_end) {
    int jB2 = B2_idx[pB2];
    int j = jB2;
    int B2_segend = pB2 + 1;
    int pA2 = (pA1 * A2_size) + j;
    A_vals[pA2] = B_vals[B2_pos];
    pB2 = B2_segend;
  }
  while (pC2 < C2_end) {
    int jC2 = C2_idx[pC2];
    int j = jC2;
    int C2_segend = pC2 + 1;
    int pA2 = (pA1 * A2_size) + j;
    A_vals[pA2] = C_vals[C2_pos];
    pC2 = C2_segend;
  }
  iB1++;
  pC1 = C1_segend;
}
```

Figure 4-7: Sparse matrix addition kernel shown in Figure 4-6 with all level function calls inlined. This version is fully specialized to CSR and COO input matrices.

Chapter 5

Evaluation

We evaluate our technique against a wide array of existing sparse linear and tensor algebra libraries and find that our technique generates sparse tensor algebra kernels for many disparate storage formats that are competitive with hand-implemented kernels in terms of performance, showing that generality and performance need not be mutually exclusive. We further find that such generality is crucial for achieving performance in real-world tensor algebra applications. Kjolstad et al. demonstrated similar claims for the technique they proposed in [34], which handles a strict subset of formats that our technique supports. As our technique generates identical code for those formats, the rest of this chapter will focus mainly on tensor formats not supported by the technique Kjolstad et al. proposed.

5.1 Experimental Setup

We implement our technique as an extension to `taco` [35], an open-source tensor algebra library that implements the compiler theory described in [34]. We evaluate our technique against six existing sparse linear algebra libraries: Intel MKL [28], SciPy [30], Eigen [27], uBLAS [72], Gmm++ [52], and OSKI [71]. Intel MKL is a math processing library for C and Fortran that is heavily optimized for Intel processors. SciPy is a widely used scientific computing library for Python. Eigen [27], uBLAS [72] and Gmm++ [52] are C++ libraries that use template metaprogramming to specialize linear algebra operations for fast

execution when possible. OSKI [71] is a C library that automatically tunes certain sparse kernels to take advantage of optimizations such as register blocking and vectorization.

We also evaluate our technique against two existing sparse tensor algebra libraries: the MATLAB Tensor Toolbox [8], and TensorFlow [1]. The Tensor Toolbox is a MATLAB library that implements a wide range of kernels and factorization algorithms for dense and sparse tensors of any order. TensorFlow is a popular machine learning library that supports a number of basic operations on sparse tensors.

All experiments are run on a two-socket, 12-core/24-thread 2.4 GHz Intel Xeon E5-2695 v2 machine with 30 MB of L3 cache per socket and 128 GB of main memory, running GCC 5.4.0 and MATLAB 2016b. We run each experiment multiple times with the cache cleared of input and output data before each run and report average execution times. All results are for single-threaded execution.

We run our experiments with matrices and higher-order tensors from real-world applications, obtained from the SuiteSparse Matrix Collection [21] and the FROSTT Tensor Collection [57]. Table 5.1 reports some relevant statistics pertaining to these tensors. We store tensor coordinates as integers whenever possible and store component values as double-precision floats; the TTM and INNERPROD kernels implemented in the Tensor Toolbox do not support integer coordinates, so for those particular two kernels we benchmark the Tensor Toolbox with double-precision floating-point indices.

5.2 Sparse Matrix Computations

Our technique emits efficient kernels that are specialized to specific computations and to tensor operands with particular attributes. As our experiments show, this lets our technique achieve competitive performance with more specialized sparse linear algebra libraries like Intel MKL that hand-implement similar kernels, while also allowing our technique to outperform more general hand-implemented libraries like TensorFlow that cannot as effectively optimize for the computation or data.

Table 5.1: Summary of tensors used in experiments. The last column describes, for each order-2 tensor (matrix), the number of diagonals that contain at least one non-zero.

Tensor	Domain	Dimensions	Non-zeros	Diagonals
pdb1HYS	Protein data base	36K × 36K	4,344,765	25,577
obstclae	Optimization	40K × 40K	197,608	5
rma10	3D CFD	46K × 46K	2,329,092	17,367
dixmaanl	Optimization	60K × 60K	299,998	7
cant	FEM/Cantilever	62K × 62K	4,007,383	99
consph	FEM/Spheres	83K × 83K	6,010,480	13,497
denormal	Counter-example prob.	89K × 89K	1,156,224	13
Baumann	Chemical master eqn.	112K × 112K	748,331	7
cop20k	FEM/Accelerator	121K × 121K	2,624,331	221,205
shipsec1	FEM	141K × 141K	3,568,176	10,475
scircuit	Circuit	171K × 171K	958,936	159,419
mac_econ	Economics	207K × 207K	1,273,389	511
pwtk	Wind tunnel	218K × 218K	11,524,432	19,929
Lin	Structural prob.	256K × 256K	1,766,400	7
synth1	Synthetic matrix	500K × 500K	1,999,996	4
synth2	Synthetic matrix	1M × 1M	1,999,999	2
ecology1	Animal movement	1M × 1M	4,996,000	5
webbase	Web connectivity	1M × 1M	3,105,536	564,259
atmosmodd	Atmospheric model	1.3M × 1.3M	8,814,880	7
Facebook	Social media	1.6K × 64K × 64K	737,934	
NELL-2	Machine learning	12K × 9.2K × 29K	76,879,419	
NELL-1	Machine learning	2.9M × 2.1M × 25M	143,599,552	

COO Kernels We measure the performance of sparse matrix-vector multiplication (SpMV), sparse matrix-dense matrix multiplication (SpDM), and sparse matrix addition kernels that compute on matrices stored in the COO format. The kernels are generated with our technique and hand-implemented in Intel MKL, SciPy, and TensorFlow. Intel MKL and SciPy only support the struct of arrays (SoA) variant of the COO format, where each tensor dimension has a corresponding array that stores only the coordinates along that dimension; this is the variant illustrated in Figure 2-2c. TensorFlow, on the other hand, only supports the array of structs (AoS) variant, where a single array stores the elements of the `idx` arrays in the SoA variant in an interleaved fashion. By contrast, our technique supports both variants, which demonstrates the versatility of our approach; supporting

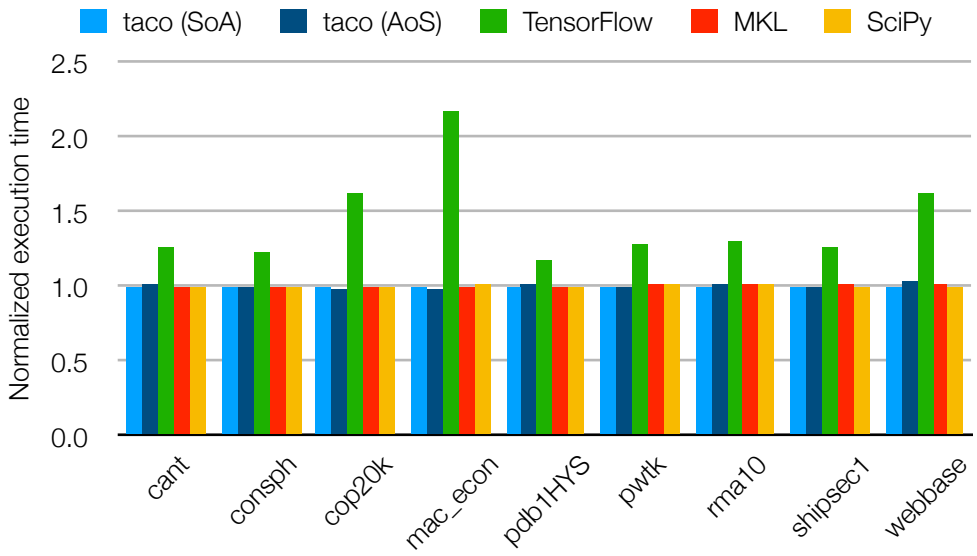


Figure 5-1: Normalized execution time of COO SpMV ($y = Ax$, where A is a COO matrix and x and y are dense vectors) with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` with SoA COO for each matrix.

the AoS variant of the COO format simply requires defining variants of the compressed and singleton level formats with the following `pos_access` function:

```
pos_access(p_k, i_1, ..., i_{k-1}):
  return <idx[2 * p_k + d], true>
```

where `d` is 0 for the row dimension and 1 for the column dimension.

Figures 5-1, 5-2, and 5-3 shows the results of this experiment. SciPy does not support computing SpDM and matrix addition directly on COO matrices and Intel MKL also does not support computing matrix addition on COO matrices, so we omit those measurements. The experimental results demonstrate that our technique emits code that is, on average, equal to or better than existing libraries in terms of performance for computations on COO matrices. In fact, the COO SpMV kernel our technique generates implements the exact same algorithm as SciPy and Intel MKL, which accounts for the identical performance between the three kernels. TensorFlow, by contrast, does not implement a dedicated COO SpMV kernel and the operation must therefore be computed with the COO SpDM kernel, casting the input vector as a matrix with a single column. Thus, TensorFlow incurs unnecessary branching overhead when computing SpMV as every access into the input

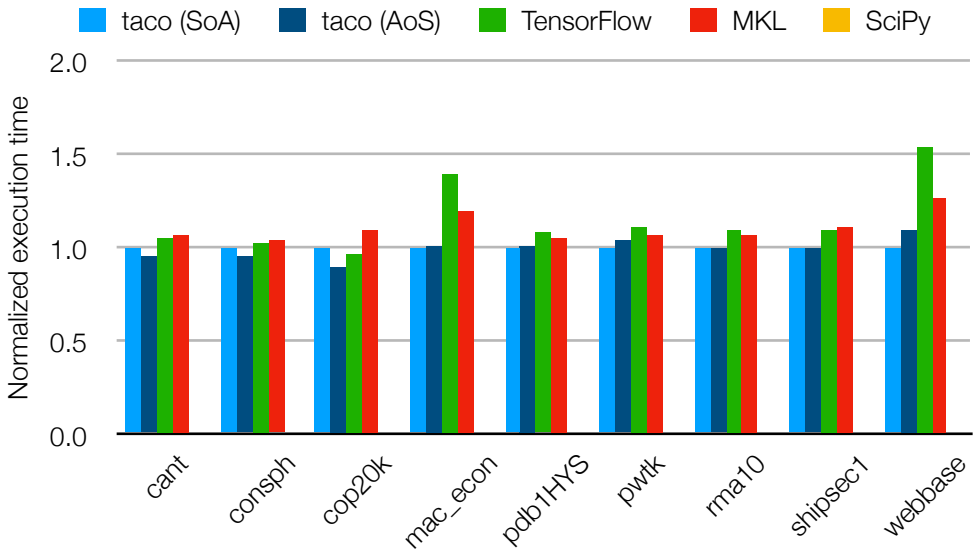


Figure 5-2: Normalized execution time of COO SpDM ($A = BC$, where B is a COO matrix and A and C are dense matrices) with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` with SoA COO for each matrix. We omit results for SciPy as it does not support computing SpDM directly on COO matrices.

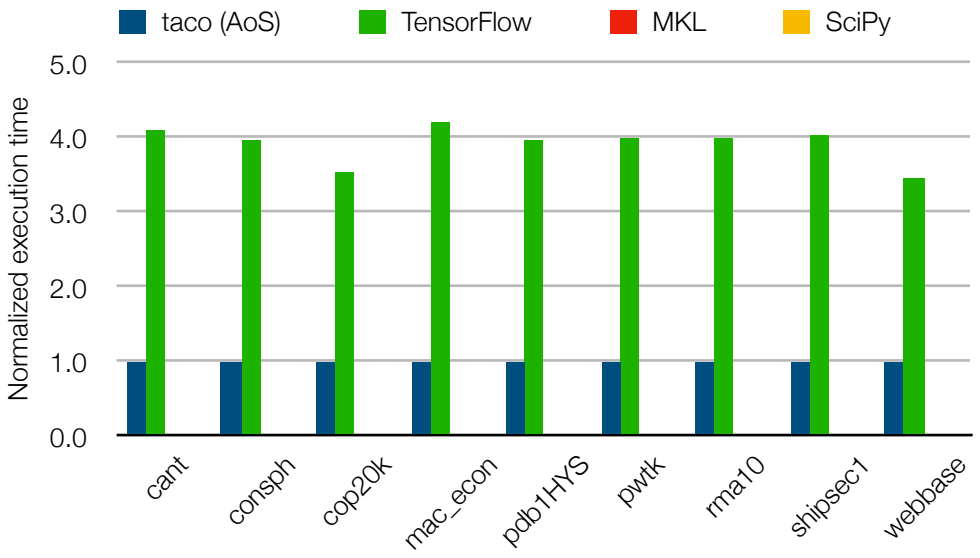


Figure 5-3: Normalized execution time of COO matrix addition ($A = B + C$, where the inputs and output are all stored in the AoS variant of the COO format) with our technique (`taco`) and TensorFlow, relative to `taco` for each matrix. We omit results for Intel MKL and SciPy as neither supports computing matrix addition directly on COO matrices.

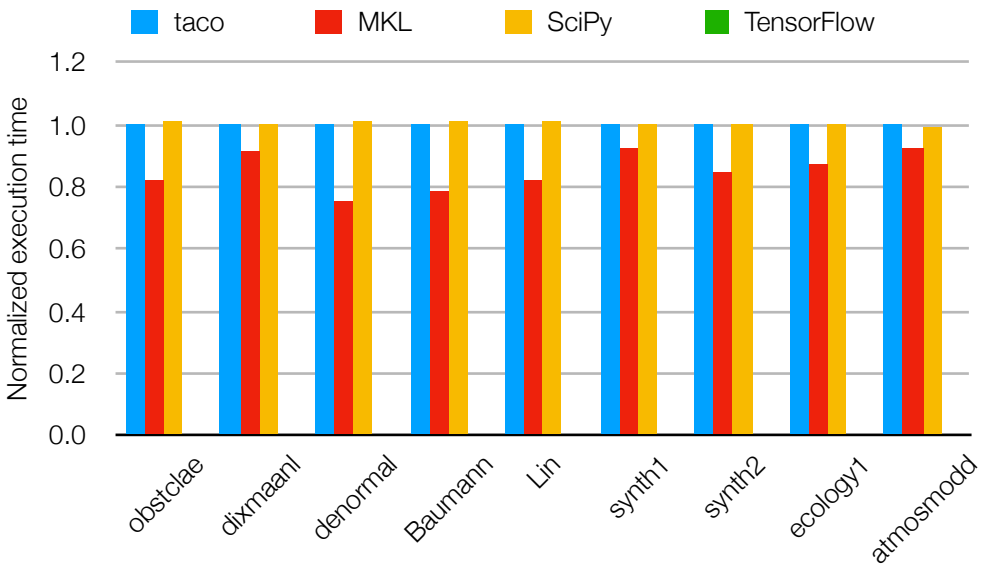


Figure 5-4: Normalized execution time of DIA SpMV with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` for each matrix. We omit results for TensorFlow as it does not support the DIA format.

vector requires a loop over its trivial column dimension. Similarly, to be able to support tensors of any order without having to implement a separate kernel for every possible order, TensorFlow's sparse tensor addition kernel requires loops that are parameterized on the tensor order for copying or comparing coordinates for a single tensor component, which introduces additional branching overhead. Our metaprogramming approach, on the other hand, generates code that is specialized to the order of the input and output tensors, which allows it to avoid unnecessary conditional branches when working with coordinates for a single tensor component.

DIA Kernels We also measure and compare the performance of SpMV kernels that compute on DIA matrices. The kernels are generated with our technique and hand-implemented in Intel MKL and SciPy; we omit TensorFlow as it does not support the DIA format. The results of this experiment are shown in Figure 5-4. Again, our technique emits code that implements the same high-level algorithm as SciPy, which explains the lack of observable performance differences between the two. Our technique is about 18% slower than Intel MKL on average as the latter tiles the computation to reduce the cache miss rate

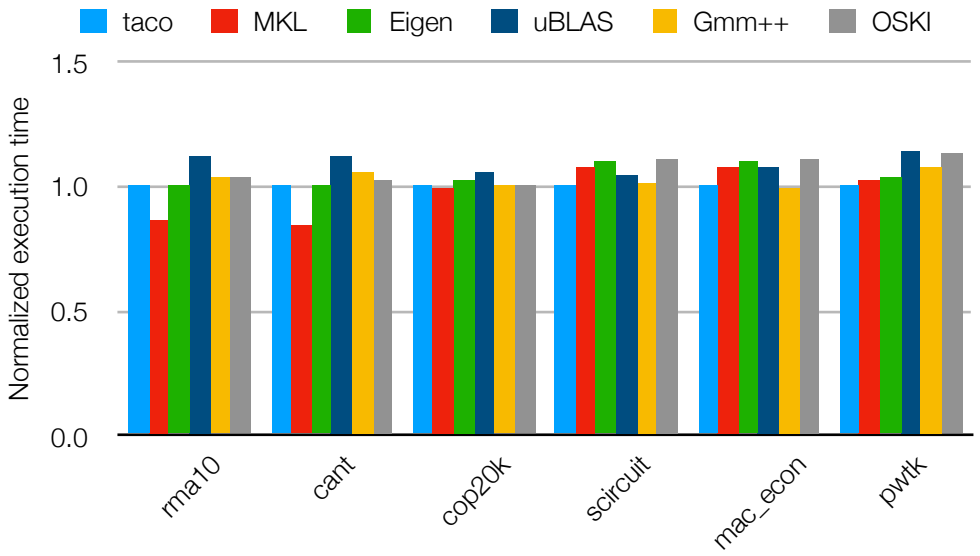


Figure 5-5: Normalized execution time of CSR SpMV with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` for each matrix.

for the input vector; future work includes generalizing our technique to support iteration space tiling, which would also be beneficial for dense kernels that take unblocked inputs.

CSR Kernels We additionally compare the performance of CSR kernels generated with our technique against existing sparse linear algebra libraries for SpMV, sampled dense-dense matrix product (SDDMM), and RESIDUAL. SDDMM has applications in machine learning [77], while RESIDUAL is used in the conjugate gradient method for instance.

Figures 5-5, 5-6, and 5-7 shows the results of this experiment. As before, we omit results for libraries that do not support a particular operation. For all three operations we benchmark, our technique emits identical code as the technique Kjolstad et al. proposed. The CSR SpMV kernel our technique generates is, on average, competitive with Intel MKL and matches, if not slightly exceeds, all the other libraries we evaluate in terms of performance. This is not very surprising as our technique emits code that implements the same high-level algorithm used by other libraries; in fact, the core of OSKI's SpMV kernel has virtually the same code structure as the kernel generated with our technique.

For SDDMM, our technique is able to minimize the number of floating-point operations needed by emitting code that exploits the locate capability provided by the dense matrix operands in order to avoid computing components of the intermediate dense matrix

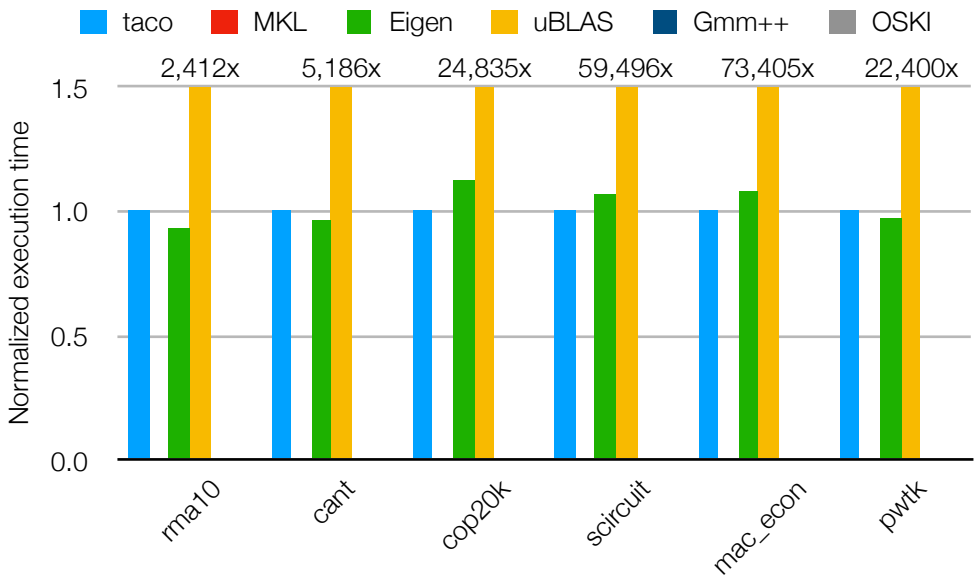


Figure 5-6: Normalized execution time of CSR SDDMM ($A = B \circ (CD)$, where A and B are CSR matrices and C and D are dense matrices) with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` for each matrix. We omit results for libraries that do not support the computation.

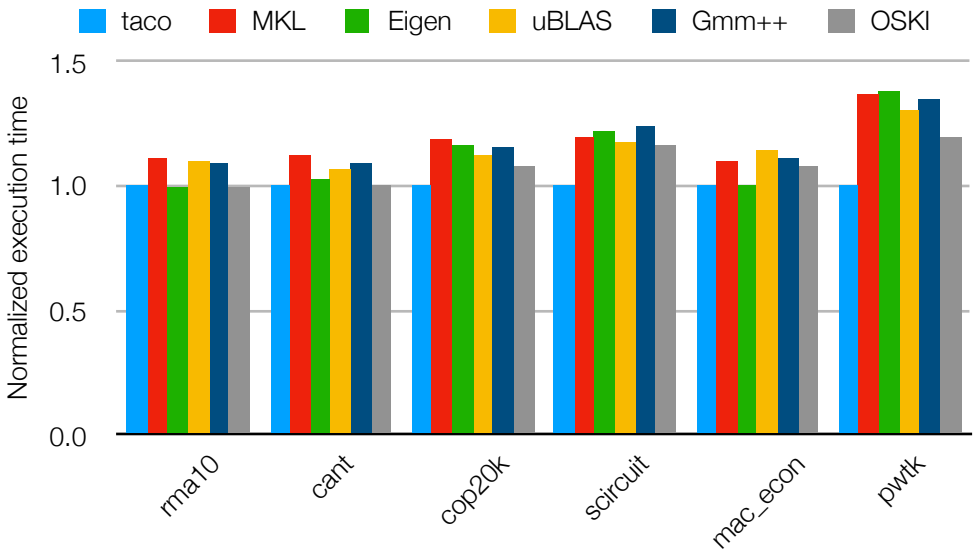


Figure 5-7: Normalized execution time of CSR RESIDUAL ($y = b - Ax$, where A is a CSR matrix and b, x, and y are dense vectors) with our technique (`taco`) and existing sparse linear algebra libraries, relative to `taco` for each matrix.

product that cannot possibly contribute non-zeros to the output (i.e., the components with coordinates that correspond to zeros in the sparse matrix operand). Eigen provides a dedicated kernel that implements the same algorithm and thus achieves similar performance. uBLAS, by contrast, effectively computes the entire dense matrix product and consequently does $\Theta(n^3)$ work as opposed to $\Theta(n \cdot \text{nnz}(B))$ for n-by-n inputs, which explains the orders of magnitude difference in performance we observe in Figure 5-6.

Our technique also generates fused compound kernels that avoid large intermediate results, which enables it to match or even exceed the performance of all the libraries we evaluate for RESIDUAL. For instance, to compute RESIDUAL with Intel MKL or OSKI, one must first copy the input vector b to the output vector—in essence treating the output as a temporary vector—and then call the SpMV kernel. The reason is that these libraries only support incrementing the output, and not some arbitrary input vector, by the result of a matrix-vector multiplication. Thus, the output vector has to be scanned twice in order to perform the computation, which results in increased memory traffic if the vector is too large to fit in cache. Our technique, on the other hand, generates code for RESIDUAL that computes each component of the matrix-vector product as needed when evaluating the top-level vector summation. This eliminates the need for a temporary vector and thereby reduces memory traffic with larger inputs such as webbase-1M.

5.3 Sparse Higher-Order Tensor Computations

We additionally compare the performance of kernels generated with our technique and equivalent hand-implemented kernels in the MATLAB Tensor Toolbox (TTB) and TensorFlow (TF) for the following higher-order tensor computations:

TTV	$A_{ij} = \sum_k B_{ijk} c_k$
TTM	$A_{ijk} = \sum_l B_{ijl} C_{kl}$
MTTKRP	$A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$
PLUS	$A_{ijk} = B_{ijk} + C_{ijk}$
INNERPROD	$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

Table 5.2: Execution time (in milliseconds) of various sparse tensor algebra kernels computing on COO tensors. Figures in parentheses show slowdown relative to our technique. A missing entry means an operation is not supported by a particular library, while OOM denotes that the kernel runs out of memory. We omit results for TensorFlow PLUS with NELL-2 and NELL-1 as TensorFlow's protocol buffers do not support tensors of those sizes.

	Facebook			NELL-2		NELL-1	
	taco	TTB	TF	taco	TTB	taco	TTB
TTV	13	149 (11.5)		350	14786 (42.2)	2302	33970 (14.8)
TTM	444	16271 (36.7)		4616	49417 (10.7)	56478	OOM
MTTKRP	44	285 (6.5)		3555	42111 (11.8)	21042	110502 (5.3)
PLUS	38	468 (12.3)	60 (1.6)	3085	72366 (23.5)	6311	123387 (19.6)
INNERPROD	10	995 (99.3)		380	144800 (381.0)	904	262605 (290.5)

where all the 3rd-order input tensors as well as the outputs of TTV, TTM, and PLUS are stored in the COO format. Here we assume all the COO tensors store non-zero coordinates in order; the sparse tensor class constructor that the Tensor Toolbox implements enforces this property, while TensorFlow's PLUS kernel also assumes this property. All of the operations we benchmark have real-world applications. TTM and MTTKRP, for example, are building blocks of widely used algorithms for computing Tucker decompositions and canonical polyadic (CP) decompositions of tensors respectively [42, 60].

Table 5.2 shows the results of this experiment.[1] The Tensor Toolbox and TensorFlow exemplify different points in the trade-off space for hand-written sparse tensor algebra libraries. The Tensor Toolbox supports all of the operations we evaluate but does not achieve high performance for any of them, while TensorFlow supports only one operation but computes it more efficiently than the Tensor Toolbox. Our technique, meanwhile, generates efficient code for computing all five operations, demonstrating that generality and performance need not be mutually exclusive.

As with sparse matrix addition, we observe that code generated specifically for adding 3rd-order COO tensors has better performance than the generic sparse tensor addition kernel in TensorFlow that can add tensors of any order. Furthermore, our technique generates code that significantly outperforms equivalent Tensor Toolbox kernels in performance, often by at least an order of magnitude. This is largely the consequence of the Tensor

[1] We omit evaluation of Intel MKL and SciPy as neither supports sparse higher-order tensor algebra.

Toolbox's general approach to sparse tensor computation, which relies on functionalities built into MATLAB that cannot always directly operate on the indices storing the tensors or exploit properties of the tensors in order to optimize the computation. To add two sparse tensors, for instance, the Tensor Toolbox computes the set of non-zero coordinates in the output by calling a MATLAB built-in function that computes the union of the sets of non-zero coordinates in the inputs. However, MATLAB's implementation of the set union operation cannot exploit the fact that the inputs are already individually sorted and therefore requires sorting the concatenation of the two input indices. By contrast, our technique emits code that is able to directly iterate over and merge the two input indices without needing to re-sort them first, which reduces the asymptotic complexity of the computation by a factor of $\Theta(\log(\text{nnz}(B) + \text{nnz}(C)))$. Additionally, our technique emits code that directly assembles sparse output indices, whereas for some computations like TTM the Tensor Toolbox has to store results in intermediate dense structures that subsequently get converted to sparse outputs.

5.4 Comparison of Formats

As we have seen, our technique generates sparse tensor kernels that are on par with hand-implemented kernels in terms of performance for a range of very different storage formats. In the rest of this section, we empirically demonstrate some of the trade-offs associated with these formats—focusing on ones not discussed in [34]—and show no tensor storage format is ideal under every circumstance.

The most suitable format for storing a tensor depends on the nature of the computation to which the tensor serves as input. The COO format, for instance, is convenient for importing and exporting sparse tensors to and from an application as it is an intuitive way to represent sparse tensors and is similar to widely used file formats for encoding sparse tensors. It is also generally not as performant as the CSR format due to its increased memory footprint. As the blue bars in Figure 5-8 show, computing matrix-vector products directly on COO matrices can take up to twice as much time as the same computation with equivalent CSR matrices. If an input matrix is imported into the application in the COO format though,

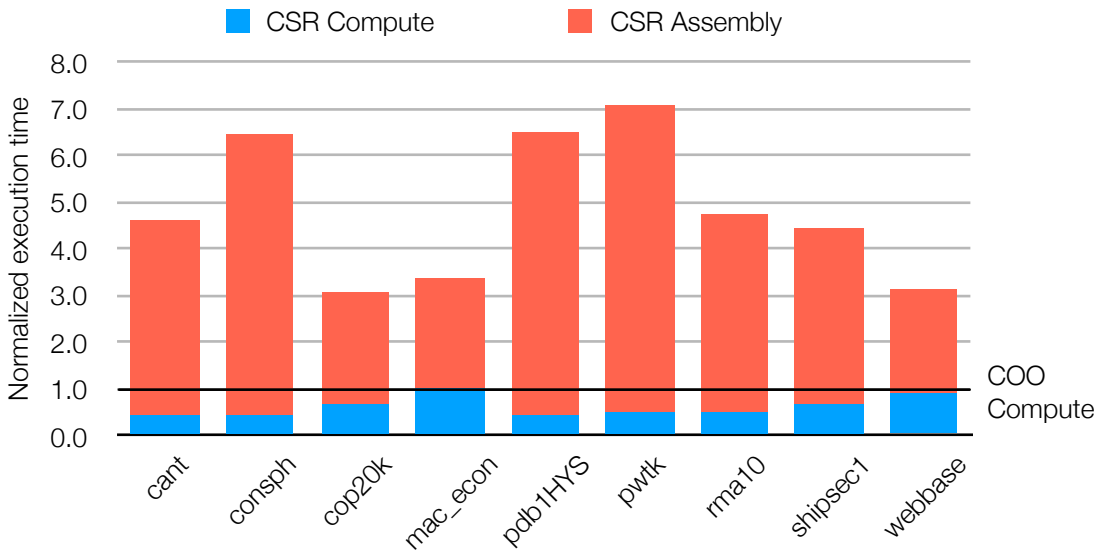


Figure 5-8: Normalized execution time of CSR SpMV relative to COO SpMV, taking into account the cost of assembling CSR indices for the input matrices. These results show that computing with CSR is faster only if the cost of assembling input matrices can be amortized over multiple computations.

then it has to be converted to a CSR matrix before the more efficient CSR SpMV kernel can be used. This preprocessing step incurs significant overhead that typically exceeds the cost of directly computing on the original COO matrix, as the red bars in Figure 5-8 show. The overhead can be amortized for iterative applications that repeatedly compute on the same matrix, making CSR SpMV worthwhile from a performance standpoint. However, for non-iterative applications, COO SpMV can offer better end-to-end performance by eliminating the need to incur format conversion overhead.

Which format provides the best performance for a particular computation also highly depends on the sparsity structure of the tensor. To illustrate this, we measure and compare the performance of SpMV computed on CSR and DIA matrices using kernels generated with our technique. As the results in Figure 5-9 demonstrate, for matrices like those benchmarked in Figure 5-4 whose non-zeros are all confined to a few densely-filled diagonals, storing them in the DIA format exposes opportunities for vectorization and can thus increase SpMV performance by up to 22% relative to CSR SpMV. On the other hand, for matrices like those benchmarked in Figure 5-1 whose non-zeros happen to be distributed among many diagonals that are all sparsely filled, the DIA format is suboptimal as it has to

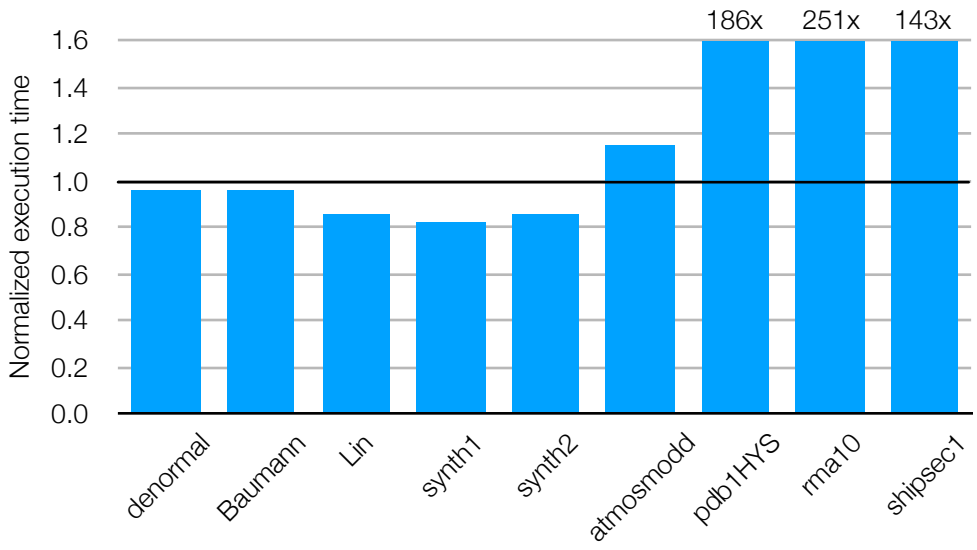


Figure 5-9: Normalized execution time of `taco`'s DIA SpMV kernel relative to `taco`'s CSR SpMV kernel. Storing the input matrix in the DIA format can accelerate SpMV if all the non-zeros in the matrix are confined to a few densely-filled diagonals, but can drastically degrade performance if that is not the case.

explicitly store all zeros in each nonempty diagonal. All these explicit zeros also have to be processed when computing the matrix-vector product even though they cannot contribute to the result. This can degrade the performance of the computation by orders of magnitude relative to CSR SpMV, as results for the rightmost three matrices in Figure 5-9 show.

Chapter 6

Related Work

Related work in this area can be roughly categorized into explorations of different sparse tensor formats (including sparse matrix and vector formats), prior work on abstractions for sparse tensor storage and code generation for sparse tensor computations, and other existing systems for sparse and dense linear and tensor algebra.

Sparse tensor formats There is a large body of work on sparse matrix and higher-order tensor formats. Sparse matrix data structures were first introduced by Tinney and Walker, who appear to have implemented the CSR data structure [68], and an early library that supports sparse operations was described by McNamee [47]. The CSC sibling is also commonly used, partly because it is convenient for direct solves [20]. Furthermore, Buluç and Gilbert proposed DCSC, which is a version of CSC that also compresses the column dimension for hypersparse matrices [16]. Many matrices also consist of dense blocks that contain most or all non-zero values and the BCSR format is designed for this [71]. Buluç et al. [15] proposed another blocked format called CSB, which can be viewed as a dense matrix consists of sparse sub-matrices, that permits efficient parallel sparse matrix-vector and matrix-transpose-vector multiplication with the same matrix representation. Buluç et al. additionally conjectured a hybrid CSB/BCSR format that can improve register reuse over CSB [15]. Researchers have also studied even more specialized formats that expose vectorization opportunities for SpMV such as ELL, which is specialized for matrices with fixed-size rows [33], and DIA, which is designed for diagonal and banded matrices [55].

Many formats for sparse higher-order tensors have also been proposed, including COO [8] and CSF [60], the latter of which generalizes (D)CSR to higher-order tensors. Baskaran et al. [10] proposed the mode-generic sparse tensor format and the mode-specific sparse tensor format, which generalize the idea of block COO to higher-order tensors for reducing memory storage and improving data locality. This work shows that all these tensor formats can be represented within the same framework with just six composable level formats that share a common interface.

Other sparse tensor formats that have been proposed include BICRS [76], which extends the CSR format to efficiently store non-zeros of a sparse matrix in Hilbert order. For efficiently computing SpMV on GPUs, Monakov et al. [48] proposed sliced ELLPACK, which generalizes ELL by partitioning the matrix into strips of a fixed number of adjacent rows, with each strip possibly storing a different number of non-zeros per row so as to minimize the number of stored zeros. Liu et al. [42] also proposed F-COO, which extends COO for enabling efficient computation of sparse higher-order tensor kernels on GPUs. Bell and Garland [11] described the HYB format, which stores most components of a matrix in an ELL submatrix and the remaining components in another COO submatrix. The HYB format is useful for computing SpMV on vector architectures with matrices that contain similar numbers of non-zeros in most rows but have a few rows that contain many more non-zeros. The Cocktail Format, which is used in clSpMV [66], generalizes HYB to support any number of submatrices that can each be stored in one of nine sparse matrix formats.

Tensor storage abstractions and code generation Researchers have explored different approaches for describing sparse vector and matrix storage for sparse linear algebra computation. Thibault et al. proposed a technique that describes regular geometric partitions in arrays and that automatically generates corresponding indexing functions [67]. This enables compression when the matrix has regular structure, but it does not generalize to general unstructured matrices.

In the context of compilers for sparse linear and tensor algebra, Kjolstad et al. proposed a formulation for tensor formats that designates each dimension as either dense or sparse, which are stored using the same physical indices as dense and compressed level types in

our abstraction [34]. However, their formulation can only describe formats that are composed strictly of those two specific types of indices, which precludes their technique from generating tensor algebra kernels that compute on many other common formats like COO and DIA. The Bernoulli project, which adopted a relational database approach to sparse linear algebra compilation, proposed a black-box protocol with access paths that describe how matrices map to physical storage [39, 65, 38]. The black-box protocol is similar to our level format interface, but they only address linear algebra and only computations involving multiplications and not additions. By contrast, the coordinate hierarchy abstraction supports code generation for any tensor algebra expression. SIPR [51], a framework that transforms dense linear algebra code to sparse code, represents sparse vectors and matrices with hard-coded element stores that provide enumerators and accessors that are analogous to level capabilities. The framework provides just two types of element store and cannot be readily extended to support new types of element store for representing other formats. Arnold [5, 4] proposed LL, a verifiable functional language for sparse matrix programs in which a sparse matrix format is defined as some nesting of lists and pairs that encode components of a dense matrix. How an LL format should be interpreted is described as part of the computation in LL, so the same computation with different matrix formats can require drastically different definitions.

Bik and Wijshoff [13, 14] developed an early compiler that transforms dense linear algebra code to equivalent sparse code by moving non-zero guards into sparse data structures. More recently, Venkat et al. [70] proposed a technique for generating inspector/executor code that may, at runtime, transform input matrices from one format to another. Both techniques support a fixed set of standard sparse matrix formats and only generate code that work with matrices stored in those formats. Finally, Rong et al. [54] proposed a technique that discovers and exploits invariant properties of matrices in a sparse linear algebra program in order to optimize the program as a whole.

Dense and sparse linear and tensor algebra systems Much work has been done on languages [29, 43, 12], libraries [3, 73, 27, 69, 56, 28], and compilers [63, 49] for dense linear algebra as well as loop transformations that can optimize dense loop nests [75, 74, 46].

For dense tensor algebra, an early effort was the Tensor Contraction Engine [6], which is a framework that automatically optimizes dense tensor contractions (multiplications) in NWChem. libtensor [24], CTF [62], and GETT [64] are all examples of systems and techniques that transform tensor contractions into dense matrix multiplications, for which many high-performance implementations exist, by transposing tensor operands. BLIS [44] avoids explicit transpositions by fusing them with later stages, while InTensLi [40] avoids transpositions altogether by computing tensor-times-matrix products in-place. Cai et al. [18] explored techniques for optimizing MTTKRP with symmetric tensors. Finally, TensorFlow [1] is a recent example of frameworks for machine learning where tensors are passed between computation kernels in a dataflow computation.

MATLAB [43], Julia [12], Eigen [27], and PETSc [9] are examples of languages, libraries, and frameworks that are popularly used for computing with sparse matrices. MATLAB, Julia, and Eigen support all basic linear algebra operations but with very few sparse matrix formats; PETSc supports distributed computations with various distributed and blocked matrix formats. OSKI [71] and pOSKI [17] are well-known sparse linear algebra libraries that support auto-tuning but are limited to computing SpMV, triangular solves, matrix powers, and simultaneously multiplying a matrix and its transpose by vectors. For sparse tensor algebra, the MATLAB Tensor Toolbox [8] is an early system that supports a wide range of computations with the COO format and other specialized formats for factorized tensors. More recently, Solomonik and Hoefler [61] described a sparse version of CTF. SPLATT [60], which supports fast shared-memory parallel MTTKRP and tensor factorizations, HyperTensor [31], which supports distributed MTTKRP, and fast shared-memory and GPU parallel sparse tensor-times-dense-matrix multiplication algorithms proposed by Li et al. [41] are all examples of techniques that avoid data transformation overhead with dedicated kernels for computing sparse tensor operations. TensorFlow similarly implements dedicated kernels to support some sparse tensor operations on COO tensors [26]. All of these approaches require sparse tensor algebra kernels to be manually implemented. By contrast, our technique automatically generates such kernels.

Chapter 7

Conclusion

We have described a new technique for generating efficient tensor algebra kernels that compute on a diverse range of storage formats. We presented a levelized hierarchical abstraction for tensor storage and showed how just six composable level formats that implement one or more common capabilities are sufficient to represent a host of widely used formats that encode tensor structures in disparate ways. We then described a code generation strategy that works purely on the aforementioned abstraction to emit optimized code, which exploits declared properties and capabilities of tensor operands without having to know how concrete level formats actually implement their capabilities. Our technique enables users to compute on tensors using storage formats that are fitted to their data.

Future work includes defining additional level formats that can be composed to enable support for more tensor formats, including custom formats designed to take advantage of specialized hardware accelerator capabilities for deep learning architectures [19] or power-law structures in social graphs for graph analytics. The code generation algorithm may also be extended with additional optimization strategies for emitting kernels that better exploit tensor operand properties for performance. For instance, the iterator fusion optimization described near the end of Section 4.5 can potentially be extended to handle merging of multiple fused iterators, which would make it applicable to operations involving multiple COO tensor operands. Additionally, the code generation algorithm may be adapted to specifically target accelerators (e.g., GPUs) and distributed memory systems (e.g., supercomputers). Our modular approach, which uses the coordinate hierarchy

abstraction to keep tensor storage formats and code generation separate, enables these lines of research to be pursued independently.

Appendix A

Sample Generated Kernels

Below are examples of matrix-vector multiplication kernels that take inputs and output stored in different combinations of formats, all of which are generated using our technique. The implementation of our technique in `taco` replaces every while loop over a single tensor dimension that contains no duplicate coordinates with a for loop.

```
for (int iA = 0; iA < A0_size; iA++) {
  int A0_pos = (0 * A0_size) + iA;
  int y0_pos = (0 * y0_size) + iA;
  double tj = 0;
  for (int jA = 0; jA < A1_size; jA++) {
    int A1_pos = (A0_pos * A1_size) + jA;
    int x0_pos = (0 * x0_size) + jA;
    tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
  }
  y_val_arr[y0_pos] = tj;
}
```

Listing A.1: Dense matrix-dense vector product with dense output

```
for (int iA = 0; iA < A0_size; iA++) {
  int A0_pos = (0 * A0_size) + iA;
  int y0_pos = (0 * y0_size) + iA;
  double tj = 0;
  for (int x0_pos = x0_pos_arr[0]; x0_pos < x0_pos_arr[1]; x0_pos++) {
```

```
    int jx = x0_idx_arr[x0_pos];
    int A1_pos = (A0_pos * A1_size) + jx;
    tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
  }
  y_val_arr[y0_pos] = tj;
}
```

Listing A.2: Dense matrix-sparse vector product with dense output

```
for (int iA = 0; iA < A0_size; iA++) {
  int A0_pos = (0 * A0_size) + iA;
  int y0_pos = (0 * y0_size) + iA;
  double tj = 0;
  for (int jA = 0; jA < A1_size; jA++) {
    int A1_pos = (A0_pos * A1_size) + jA;
    int x0_pos = jA % x0_width;
    if (x0_idx_arr[x0_pos] != jA && x0_idx_arr[x0_pos] != -1) {
      int end = x0_pos;
      do {
        x0_pos = (x0_pos + 1) % x0_width;
      } while (x0_idx_arr[x0_pos] != jA &&
               x0_idx_arr[x0_pos] != -1 && x0_pos != end);
    }
    if (x0_idx_arr[x0_pos] == jA) {
      tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
    }
  }
  y_val_arr[y0_pos] = tj;
}
```

Listing A.3: Dense matrix-hash map vector product with dense output

```
for (int A0_pos = A0_pos_arr[0]; A0_pos < A0_pos_arr[1]; A0_pos++) {
  int iA = A0_idx_arr[A0_pos];
  int y0_pos = (0 * y0_size) + iA;
  int A1_pos = A0_pos;
  int jA = A1_idx_arr[A1_pos];
  int x0_pos = (0 * x0_size) + jA;
  double tj = A_val_arr[A1_pos] * x_val_arr[x0_pos];
```

```
  y_val_arr[y0_pos] = y_val_arr[y0_pos] + tj;
}
```

Listing A.4: COO matrix-dense vector product with dense output

```
int A0_pos = A0_pos_arr[0];
while (A0_pos < A0_pos_arr[1]) {
  int iA = A0_idx_arr[A0_pos];
  int y0_pos = (0 * y0_size) + iA;
  int A0_segend = A0_pos + 1;
  while ((A0_segend < A0_pos_arr[1]) && (A0_idx_arr[A0_segend] == iA)) {
    A0_segend++;
  }
  double tj = 0;
  int A1_pos = A0_pos;
  while (A1_pos < A0_segend) {
    int jA = A1_idx_arr[A1_pos];
    int x0_pos = (0 * x0_size) + jA;
    double A1_val = A_val_arr[A1_pos];
    int A1_segend = A1_pos + 1;
    while ((A1_segend < A0_segend) && (A1_idx_arr[A1_segend] == jA)) {
      A1_val += A_val_arr[A1_segend];
      A1_segend++;
    }
    tj += A1_val * x_val_arr[x0_pos];
    A1_pos = A1_segend;
  }
  y_val_arr[y0_pos] = tj;
  A0_pos = A0_segend;
}
```

Listing A.5: COO matrix-dense vector product with dense output and duplicates

```
int y0_pos = 0;
int A0_pos = A0_pos_arr[0];
while (A0_pos < A0_pos_arr[1]) {
  int iA = A0_idx_arr[A0_pos];
  int A0_segend = A0_pos + 1;
  while ((A0_segend < A0_pos_arr[1]) && (A0_idx_arr[A0_segend] == iA)) {
```

```
    A0_segend++;
  }
  double tj = 0;
  int A1_pos = A0_pos;
  int x0_pos = x0_pos_arr[0];
  while ((A1_pos < A0_segend) && (x0_pos < x0_pos_arr[1])) {
    int jA = A1_idx_arr[A1_pos];
    int jx = x0_idx_arr[x0_pos];
    int j = min(jA, jx);
    if ((jA == j) && (jx == j)) {
      tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
    }
    if (jA == j) A1_pos++;
    if (jx == j) x0_pos++;
  }
  y_val_arr[y0_pos] = tj;
  y0_idx_arr[y0_pos] = iA;
  y0_pos++;
  A0_pos = A0_segend;
}
y0_pos_arr[1] = y0_pos;
```

Listing A.6: COO matrix-sparse vector product with sparse output

```
for (int iA = 0; iA < A0_size; iA++) {
  int A0_pos = (0 * A0_size) + iA;
  int y0_pos = (0 * y0_size) + iA;
  double tj = 0;
  for (int A1_pos = A1_pos_arr[A0_pos];
           A1_pos < A1_pos_arr[A0_pos + 1]; A1_pos++) {
    int jA = A1_idx_arr[A1_pos];
    int x0_pos = (0 * x0_size) + jA;
    tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
  }
  y_val_arr[y0_pos] = tj;
}
```

Listing A.7: CSR matrix-dense vector product with dense output

```
for (int iA = 0; iA < A0_size; iA++) {
  int A0_pos = (0 * A0_size) + iA;
  int y0_pos = (0 * y0_size) + iA;
  double tj = 0;
  int A1_pos = A1_pos_arr[A0_pos];
  int x0_pos = x0_pos_arr[0];
  while ((A1_pos < A1_pos_arr[A0_pos + 1]) && (x0_pos < x0_pos_arr[1])) {
    int jA = A1_idx_arr[A1_pos];
    int jx = x0_idx_arr[x0_pos];
    int j = min(jA, jx);
    if ((jA == j) && (jx == j)) {
      tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
    }
    if (jA == j) A1_pos++;
    if (jx == j) x0_pos++;
  }
  y_val_arr[y0_pos] = tj;
}
```

Listing A.8: CSR matrix-sparse vector product with dense output

```
for (int jA = 0; jA < A0_size; jA++) {
  int A0_pos = (0 * A0_size) + jA;
  int x0_pos = (0 * x0_size) + jA;
  double tj = x_val_arr[x0_pos];
  for (int A1_pos = A1_pos_arr[A0_pos];
           A1_pos < A1_pos_arr[A0_pos + 1]; A1_pos++) {
    int iA = A1_idx_arr[A1_pos];
    int y0_pos = (0 * y0_size) + iA;
    y_val_arr[y0_pos] = y_val_arr[y0_pos] + (A_val_arr[A1_pos] * tj);
  }
}
```

Listing A.9: CSC matrix-dense vector product with dense output

```
for (int x0_pos = x0_pos_arr[0]; x0_pos < x0_pos_arr[1]; x0_pos++) {
  int jx = x0_idx_arr[x0_pos];
  int A0_pos = (0 * A0_size) + jx;
```

```
  double tj = x_val_arr[x0_pos];
  for (int A1_pos = A1_pos_arr[A0_pos];
           A1_pos < A1_pos_arr[A0_pos + 1]; A1_pos++) {
    int iA = A1_idx_arr[A1_pos];
    int y0_pos = (0 * y0_size) + iA;
    y_val_arr[y0_pos] = y_val_arr[y0_pos] + (A_val_arr[A1_pos] * tj);
  }
}
```

Listing A.10: CSC matrix-sparse vector product with dense output

```
int y0_pos = 0;
for (int A0_pos = A0_pos_arr[0]; A0_pos < A0_pos_arr[1]; A0_pos++) {
  int iA = A0_idx_arr[A0_pos];
  double tj = 0;
  for (int A1_pos = A1_pos_arr[A0_pos];
           A1_pos < A1_pos_arr[A0_pos + 1]; A1_pos++) {
    int jA = A1_idx_arr[A1_pos];
    int x0_pos = (0 * x0_size) + jA;
    tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
  }
  y_val_arr[y0_pos] = tj;
  y0_idx_arr[y0_pos] = iA;
  y0_pos++;
}
y0_pos_arr[1] = y0_pos;
```

Listing A.11: DCSR matrix-dense vector product with sparse output

```
int y0_pos = 0;
for (int A0_pos = A0_pos_arr[0]; A0_pos < A0_pos_arr[1]; A0_pos++) {
  int iA = A0_idx_arr[A0_pos];
  double tj = 0;
  int A1_pos = A1_pos_arr[A0_pos];
  int x0_pos = x0_pos_arr[0];
  while ((A1_pos < A1_pos_arr[A0_pos + 1]) && (x0_pos < x0_pos_arr[1])) {
    int jA = A1_idx_arr[A1_pos];
    int jx = x0_idx_arr[x0_pos];
    int j = min(jA, jx);
```

```
    if ((jA == j) && (jx == j)) {
      tj += A_val_arr[A1_pos] * x_val_arr[x0_pos];
    }
    if (jA == j) A1_pos++;
    if (jx == j) x0_pos++;
  }
  y_val_arr[y0_pos] = tj;
  y0_idx_arr[y0_pos] = iA;
  y0_pos++;
}
y0_pos_arr[1] = y0_pos;
```

Listing A.12: DCSR matrix-sparse vector product with sparse output

```
for (int kA = 0; kA < A0_size; kA++) {
  int A0_pos = (0 * A0_size) + kA;
  for (int iA = 0; iA < A1_size; iA++) {
    int A1_pos = (A0_pos * A1_size) + iA;
    int y0_pos = (0 * y0_size) + iA;
    int A2_pos = A1_pos;
    int jA = A2_idx_arr[A2_pos];
    int x0_pos = (0 * x0_size) + jA;
    double tj = A_val_arr[A2_pos] * x_val_arr[x0_pos];
    y_val_arr[y0_pos] = y_val_arr[y0_pos] + tj;
  }
}
```

Listing A.13: ELL matrix-dense vector product with dense output

```
for (int kA = 0; kA < A0_size; kA++) {
  int A0_pos = (0 * A0_size) + kA;
  for (int iA = max(0, -A1_offset[kA]);
           iA < min(A1_size, A2_size - A1_offset[kA]); iA++) {
    int A1_pos = (A0_pos * A1_size) + iA;
    int y0_pos = (0 * y0_size) + iA;
    int A2_pos = A1_pos;
    int jA = iA + A1_offset[kA];
    int x0_pos = (0 * x0_size) + jA;
    double tj = A_val_arr[A2_pos] * x_val_arr[x0_pos];
```

```
    y_val_arr[y0_pos] = y_val_arr[y0_pos] + tj;
  }
}
```

Listing A.14: DIA matrix-dense vector product with dense output

```
for (int i1A = 0; i1A < A0_size; i1A++) {
  int A0_pos = (0 * A0_size) + i1A;
  int y0_pos = (0 * y0_size) + i1A;
  for (int A1_pos = A1_pos_arr[A0_pos];
           A1_pos < A1_pos_arr[A0_pos + 1]; A1_pos++) {
    int j1A = A1_idx_arr[A1_pos];
    int x0_pos = (0 * x0_size) + j1A;
    for (int i2A = 0; i2A < A2_size; i2A++) {
      int A2_pos = (A1_pos * A2_size) + i2A;
      int y1_pos = (y0_pos * y1_size) + i2A;
      double tj2 = 0;
      for (int j2A = 0; j2A < A3_size; j2A++) {
        int A3_pos = (A2_pos * A3_size) + j2A;
        int x1_pos = (x0_pos * x1_size) + j2A;
        tj2 += A_val_arr[A3_pos] * x_val_arr[x1_pos];
      }
      y_val_arr[y1_pos] = y_val_arr[y1_pos] + tj2;
    }
  }
}
```

Listing A.15: BCSR matrix-dense vector product with dense output

```
for (int i1A = 0; i1A < A0_size; i1A++) {
  int A0_pos = (0 * A0_size) + i1A;
  int y0_pos = (0 * y0_size) + i1A;
  for (int j1A = 0; j1A < A1_size; j1A++) {
    int A1_pos = (A0_pos * A1_size) + j1A;
    int x0_pos = (0 * x0_size) + j1A;
    for (int A2_pos = A2_pos_arr[A1_pos];
             A2_pos < A2_pos_arr[A1_pos + 1]; A2_pos++) {
      int i2A = A2_idx_arr[A2_pos];
      int y1_pos = (y0_pos * y1_size) + i2A;
```

```
      int A3_pos = A2_pos;
      int j2A = A3_idx_arr[A3_pos];
      int x1_pos = (x0_pos * x1_size) + j2A;
      double tj2 = A_val_arr[A3_pos] * x_val_arr[x1_pos];
      y_val_arr[y1_pos] = y_val_arr[y1_pos] + tj2;
    }
  }
}
```

Listing A.16: CSB matrix-dense vector product with dense output

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[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] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

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

[5] 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 (ICFP '10)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1863543.1863581

[6] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. 2006. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics* 104, 2 (2006), 211–228.

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

[8] 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.

[9] Satish Balay, William D Gropp, Lois Curfman McInnes, and Barry F Smith. 1997. Efficient management of parallelism in object-oriented numerical software libraries. In *Modern software tools for scientific computing*. Springer, Birkhäuser Boston, 163–202.

[10] 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

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

[12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. (2012).

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Jong-Ho Byun, Richard Lin, Katherine A Yelick, and James Demmel. 2012. Autotuning sparse matrix-vector multiplication for multicore. *EECS, UC Berkeley, Tech. Rep* (2012).

[18] Jonathon Cai, Muthu Baskaran, Benoît Meister, and Richard Lethin. 2015. Optimization of symmetric tensor computations. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 1–7.

[19] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan 2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[20] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.

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

[22] 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 (ICCS'05)*. Springer-Verlag, Berlin, Heidelberg, 99–106. https://doi.org/10.1007/11428831_13

[23] Albert. Einstein. 1916. The Foundation of the General Theory of Relativity. *Annalen der Physik* 354 (1916), 769–822.

[24] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I Krylov. 2013. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry* 34, 26 (2013), 2293–2309.

[25] Richard Feynman, Robert B. Leighton, and Matthew L. Sands. 1963. *The Feynman Lectures on Physics*. Vol. 3. Addison-Wesley.

[26] Google. 2017. TensorFlow Sparse Tensors. https://www.tensorflow.org/api_guides/python/sparse_ops. (2017).

[27] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org. (2010).

[28] Intel. 2012. *Intel math kernel library reference manual*. Technical Report. 630813-051US, 2012. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf.

[29] Kenneth E. Iverson. 1962. *A Programming Language*. Wiley.

[30] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/ [Online; accessed <today>].

[31] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 77.

[32] Venera Khoromskaia and Boris N. Khoromskij. 2015. Tensor numerical methods in quantum chemistry: from Hartree-Fock to excitation energies. *Phys. Chem. Chem. Phys.* 17 (2015), 31491–31509. Issue 47. https://doi.org/10.1039/C5CP01215E

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

[34] 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

[35] Fredrik Kjolstad, David Lugato, Stephen Chou, Shoaib Kamil, et al. 2017. The Tensor Algebra Compiler. https://github.com/tensor-compiler/taco. (2017).

[36] Donald Ervin Knuth. 1973. *The art of computer programming: sorting and searching*. Vol. 3. Pearson Education.

[37] Joseph C Kolecki. 2002. An Introduction to Tensors for Students of Physics and Engineering. *Unixenguaedu* 7, September (2002), 29.

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

[39] 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.

[40] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 76.

[41] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. 2016. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 26–33.

[42] 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

[43] MATLAB. 2014. *version 8.3.0 (R2014a)*. The MathWorks Inc., Natick, Massachusetts.

[44] Devin Matthews. 2017. *High-Performance Tensor Contraction without Transposition*. Technical Report.

[45] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 165–172.

[46] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18, 4 (1996), 424–453.

[47] John Michael McNamee. 1971. Algorithm 408: a sparse matrix package (part I)[F4]. *Commun. ACM* 14, 4 (1971), 265–273.

[48] 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.

[49] Thomas Nelson, Geoffrey Belter, Jeremy G. Siek, Elizabeth Jessup, and Boyana Norris. 2015. Reliable Generation of High-Performance Matrix Algebra. *ACM Trans. Math. Softw.* 41, 3, Article 18 (June 2015), 27 pages.

[50] National Institute of Standards and Technology. 2013. Matrix Market: File Formats. (14 August 2013). http://math.nist.gov/MatrixMarket/formats.html

[51] 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.

[52] Yves Renard. 2017. Gmm++. (2017). http://download.gna.org/getfem/html/homepage/gmm/first-step.html

[53] Gregorio Ricci-Curbastro and Tullio Levi-Civita. 1901. Méthodes de calcul différentiel absolu et leurs applications. *Math. Ann.* 54 (1901).

[54] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 247–259.

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

[56] Conrad Sanderson. 2010. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Technical Report. NICTA.

[57] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. (2017). http://frostt.io/

[58] 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.

[59] Shaden Smith, Jongsoo Park, and George Karypis. 2017. Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory. *31st IEEE International Parallel & Distributed Processing Symposium (IPDPS'17)* (2017).

[60] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 61–70.

[61] Edgar Solomonik and Torsten Hoefler. 2015. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv preprint arXiv:1512.00066* (2015).

[62] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.

[63] Daniele G Spampinato and Markus Püschel. 2014. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 23.

[64] Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *arXiv preprint arXiv:1607.00145* (2016).

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

[66] 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 (ICS '12)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/2304576.2304624

[67] Scott Thibault, Lenore Mullin, and Matt Insall. 1994. Generating Indexing Functions of Regularly Sparse Arrays for Array Compilers. (1994).

[68] 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.

[69] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[70] 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 (PLDI 2015)*. 521–532.

[71] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 1 (2005), 521+.

[72] Joerg Walter and Mathias Koch. 2007. uBLAS. (2007). http://www.boost.org/libs/numeric/ublas/doc/index.htm

[73] R. Clint Whaley and Jack Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*.

[74] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. *SIGPLAN Not.* 26, 6 (May 1991), 30–44.

[75] Michael Joseph Wolfe. 1982. *Optimizing Supercompilers for Supercomputers*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL, USA. AAI8303027.

[76] 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.

[77] Huasha Zhao. 2014. *High Performance Machine Learning through Codesign and Rooflining*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.