Format Abstractions for the Compilation of Sparse Tensor Algebra

by

Stephen Chou

BASc, University of Waterloo (2015)
SM, Massachusetts Institute of Technology (2018)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 26, 2022

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

Format Abstractions for the Compilation of Sparse Tensor Algebra

by

Stephen Chou

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Tensors are commonly used to represent data in many domains, including data analytics, machine learning, science, and engineering. Many highly-optimized libraries and compilers have been developed for efficiently computing on dense tensors. However, existing libraries and compilers are limited in their ability to support real-world applications that work with sparse tensors, which contain mostly zeros. In particular, there exist countless specialized formats for storing sparse tensors in memory, each suited to specific types of applications and data. Since different formats often use very different data structures to store nonzeros though, computing with sparse tensors that are stored in different formats can require vastly dissimilar code that are all difficult to implement by hand and non-trivial to generate automatically. Existing libraries and compilers must therefore limit the set of computations and formats that they directly support, sacrificing usability and performance as a result.

In this dissertation, I describe how to build a compiler that supports efficiently computing on sparse tensors that may be stored in a wide variety of formats. I first show how many commonly-used sparse tensor formats—from array-based formats like CSR, COO, and DIA to formats that store nonzeros using pointer-based data structures like linked lists, BSTs, and C-trees—can all be expressed as compositions of per-dimension formats. I further show how such per-dimension formats can be precisely defined by implementing a common set of abstractions that capture how their underlying data structures store nonzeros in memory and that capture how these data structures can be efficiently accessed or constructed. I then demonstrate how, with such specifications of per-dimension formats at hand, a compiler can generate code to efficiently compute on tensors that are stored in any of the aforementioned—and countless other—formats. We have implemented our technique in the TACO sparse tensor algebra compiler, which is the first compiler to generate code that computes any basic tensor algebra expression with sparse tensors that may be stored in arbitrary formats. Our technique generates code that has performance competitive with, if not better than, equivalent code in hand-optimized libraries and frameworks.

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

Acknowledgments

I would like to thank Saman Amarasinghe, who advised and mentored me over the past seven years. Throughout this time, he has provided me with much invaluable advice, both regarding research and regarding life more generally. He always encouraged me to think big and constantly pushed me to improve myself. At the same time, he was always very understanding and supportive, never hesitating whenever I needed help with anything.

I would also like to thank Julian Shun and Alan Edelman, who served on my thesis committee and provided me with valuable feedback on my thesis. Additionally, I would like to thank Michael Carbin and Martin Rinard, who offered me the opportunity to serve as teaching assistant for the Dynamic Computer Language Engineering course and helped me gain invaluable insight into how academia works.

This dissertation incorporates work that was originally published (or was conditionally accepted) in [26, 27, 25, 54, 42, 33] and that was also described in part in [24]. Special thanks goes to Fredrik Kjolstad, who initiated the TACO project and was a co-author of many of these publications. I have been working with him almost as soon as I started my graduate studies, and over the past seven years I have learned much from him regarding how to do great research. I would like to further acknowledge Saman, Shoaib Kamil, David Lugato, Daniel Donenfeld, Rawn Henry, Olivia Hsu, Rohan Yadav, and Kunle Olukotun, who were also fellow co-authors of the aforementioned publications. Additionally, I would like to acknowledge Suzanne Mueller, Willow Ahrens, Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, and Jessica Shi, who I have had the pleasure of working with on other publications not cited above.

Furthermore, I would like to thank everyone in the COMMIT group for having made my experience at MIT as enjoyable as it was. In particular, I am grateful for all the interesting discussions I had with Fred, Willow, Charith Mendis, Yunming Zhang, Riyadh Baghdadi, Tom Chen, and Ajay Brahmakshatriya. I also enjoyed all the exciting pool games that Fred, Willow, Charith, Tom, and I got to play together. Additionally, I am grateful for all the help and advice Yunming gave me about life after graduate school. Finally, I would like to thank Mary McDavitt for all her help with administrative matters over the past seven years.

Most of all, I would like to thank my parents, Hui Fen Chou-Wu and C. Perry Chou, and my wife, Yaqing Xu, for their unconditional love and support. My parents have sacrificed much to provide me with everything I ever needed, and without them I could not possibly be here. I am also eternally grateful to have been able to meet and marry Yaqing, whose constant encouragement and support helped me get to where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tensors (i.e., multidimensional arrays) are commonly used to represent data in many domains, including data analytics [9], machine learning [88, 80], science [35, 37, 51], and engineering [57]. As far back as 1957, Fortran [8] has provided first-class support for dense arrays, and since then countless more systems have been developed that let users easily and efficiently compute on dense tensors. Array programming languages such as APL [45], MATLAB [69], and modern variants of Fortran provide the ability to operate on dense tensors using scalar operators. Meanwhile, linear algebra libraries like Intel MKL [44] and Eigen [39] as well as domain-specific compilers like Halide [87] and the Tensor Contraction Engine [7] provide similar capabilities for other programming languages. These systems support a wide range of computations, and they can also optimize the running times of these computations by employing a myriad of sophisticated techniques, such as cache blocking, parallel execution, and even fusion of sub-operations.

However, many real-world applications actually work with *sparse* tensors, which are tensors that contain mostly zero elements. A graph that encodes friendship relations in a social network, for instance, can be represented by a tensor (i.e., its adjacency matrix), and such a tensor would likely be sparse since most people only have a limited number of friends. As we will see, sparse tensors like this can be stored in *formats* (i.e., data layouts) that compress out the zero elements. This not only reduces the amount of memory needed to store sparse tensors but also reduces the cost of computing on sparse tensors by asymptotic factors, since applications can avoid computing with the zero elements.

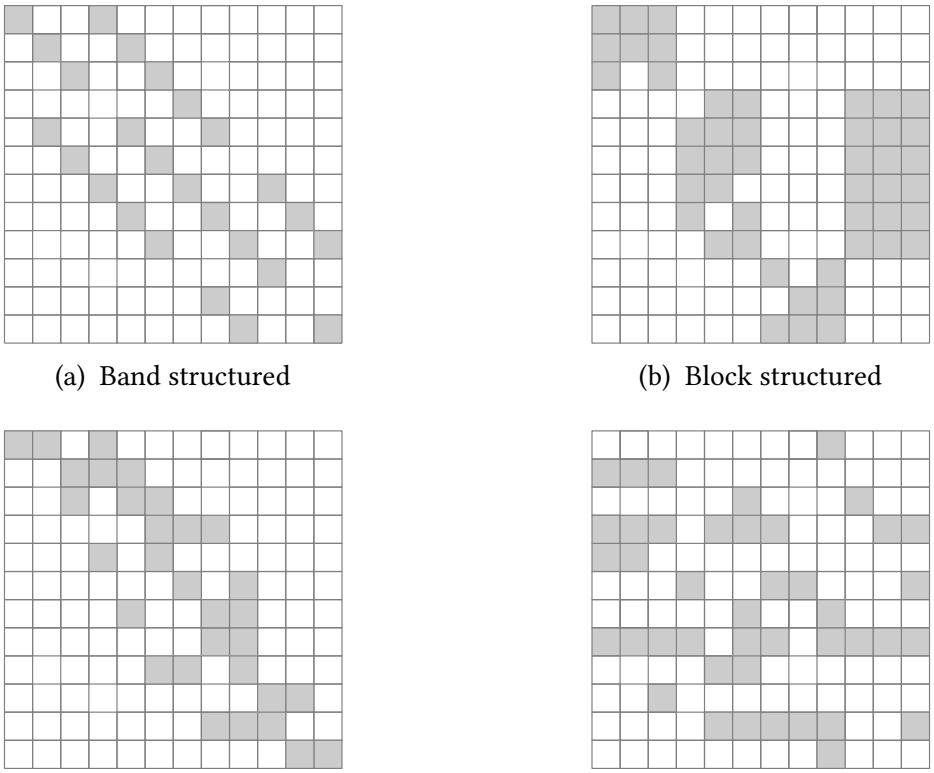
Unfortunately, existing systems for computing on sparse tensors are much more limited in their capabilities. Some of these systems significantly restrict the set of computations that are supported, thereby limiting the set of applications that can be implemented with those systems. Other systems, meanwhile, support a richer set of computations but incur significant performance penalties when performing these computations, which waste computing resources and limit the sizes of problems that can be solved with those systems. As it turns out, such limitations exist because applications in different domains often have to work with data that exhibit different types of sparsity, but it is challenging to build systems that can fully support and optimize for all these different types of sparse data.

In this dissertation, I demonstrate it is possible to build systems that do not have to sacrifice generality or performance in order to compute on sparse tensors. In particular, I show how a compiler can generate efficient code to compute arbitrary tensor algebra operations on sparse tensors that are stored in disparate formats optimized for different types of sparsity. As we will see, our compiler-based approach effectively amortizes the programmer effort needed to support efficiently working with many types of sparse data, thereby making it possible to avoid trade-offs that are required by hand-optimized systems.

1.1 Diversity of Sparsity

What makes dense tensors easier to work with is that tensors with the same dimensions must share the same set of elements that are non-zero, since all elements are non-zero by definition. This means the same optimized code can be used to efficiently compute on all dense tensors with the same dimensions. The same is not true of sparse tensors, however, since sparse tensors from different domains often differ significantly in how sparse they are, how nonzeros are distributed, and how the set of nonzeros evolves over time.

Degree of sparsity Sparse tensors that represent data from different domains often contain significantly different proportions of elements that are non-zero. Tensors that encode weights in a pruned neural network layer, for instance, typically have densities (i.e., the ratio of nonzeros to all elements in a tensor) that are on the order of 10–50%. By



(a) Band structured

(b) Block structured

(c) Bounded-degree structured

(d) Power-law structured

Figure 1-1: Examples of sparse tensors that exhibit different types of sparsity structure.

contrast, tensors that arise from finite element simulations are typically several orders of magnitude sparser, having densities that are on the order of 0.01–0.1%. Meanwhile, tensors that encode interactions on social networks or hyperlinks between web pages tend to be even sparser, with densities that can be as low as on the order of 10^{-8}%.

Sparsity structure Sparse tensors that represent data from different domains often also have very different sparsity structures. Tensors that encode stencils applied to regular grids, for instance, typically have their nonzeros clustered along a few densely-filled diagonals, as illustrated in Figure 1-1a. Tensors that encode sparse attention mechanisms in transformer models for natural language processing, meanwhile, often have their nonzeros clustered in densely-filled blocks, as illustrated in Figure 1-1b. By contrast, tensors that represent real-world social networks, road networks, or web graphs generally do not exhibit any regular structure, though even such unstructured sparse tensors can differ significantly in how nonzeros are distributed. Tensors that represent road networks, for example,

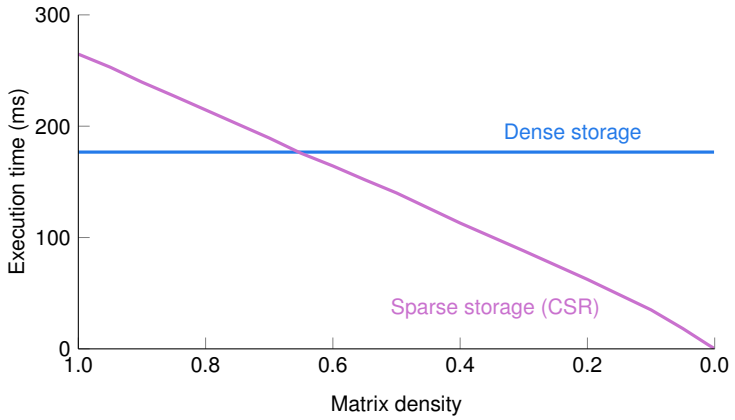


Figure 1-2: Performance of matrix-vector multiplication with the matrix stored in dense storage and in sparse storage.

typically contain similar numbers of nonzeros in every row (since intersections are typically connected to similar numbers of roads), as illustrated in Figure 1-1c. Meanwhile, tensors that represent social networks or web graphs are often power-law distributed, with a small number of rows (which might represent celebrities or popular websites) containing many nonzeros and a large number of rows (representing average users or websites) containing few nonzeros, as illustrated in Figure 1-1d.

Dynamism Applications that work with sparse tensors can further differ in how frequently they modify the tensors' sparsity structures during program execution. Finite element simulations, for instance, generally assemble sparse tensors in one shot and do not modify them afterwards. By contrast, applications that continuously analyze social networks in real time typically have to modify the sparse tensor representations of those social networks periodically in order to reflect updates (e.g., inserting new nonzeros to reflect new interactions between users).

1.2 Exploiting Sparsity for Performance

One can efficiently work with all these different types of sparse data by storing them in formats that are optimized for specific characteristics of the data. As an example, one of the most commonly used formats for storing sparse matrices (i.e., two-dimensional tensors) is

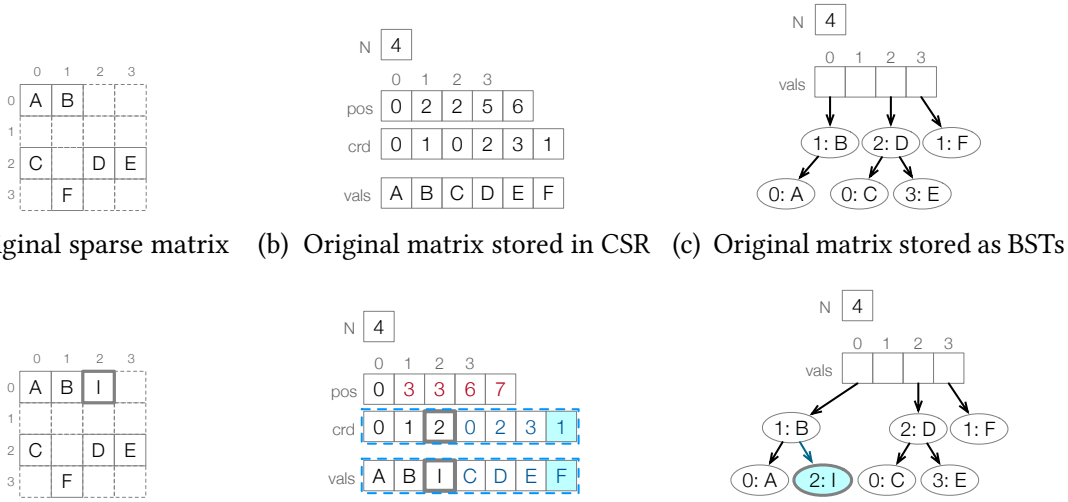
the compressed sparse row (CSR) format, which explicitly stores the column coordinates and values of nonzeros while using an auxiliary array to track which nonzeros belong to each row. The CSR format actually requires more memory to store each nonzero than dense arrays, since dense arrays do not need to explicit store the coordinates of nonzeros. However, since CSR only has to store nonzeros while dense arrays must also explicitly store zeros, CSR can exploit the fact that tensors from many domains are often extremely sparse in order to reduce the total amount of memory needed to store the tensors. This, in turn, improves the performance of computations on those same sparse tensors by reducing memory traffic at run time. In fact, as Figure 1-2 illustrates, improvements in performance that come from exploiting sparsity is typically asymptotic and becomes more significant as the sparsity of the tensor increases.

However, there also exist many other sparse tensor formats that are further optimized for storing tensors with specific sparsity structures. The diagonal (DIA) format, for instance, is optimized for sparse matrices that have nonzeros clustered along densely-filled diagonals. The block compressed sparse row (BCSR) format, on the other hand, is optimized for sparse matrices that have nonzeros clustered in densely-filled blocks of a fixed size. The ELLPACK (ELL) format, meanwhile, is optimized for sparse matrices that contain similar numbers of nonzeros in each row. Such structured sparse tensor formats exploit patterns in how nonzeros are distributed within a tensor in order to minimize the amount of auxiliary data (e.g., coordinates of nonzeros) that need to be explicitly stored as well as reduce the amount of indirection needed to access stored nonzeros. This can further improve compute performance relative to storing the matrix in CSR. That said, this potential for improvements in compute performance comes with two major caveats. First, such performance improvements can only be realized if a tensor actually exhibits the same type of sparsity structure as what the format used to store the tensor is designed for. In fact, as we will see, using a sparse tensor format that is designed for a certain type of sparsity structure to store a tensor that exhibits a different type of sparsity structure (e.g., storing an power-law structured sparse matrix in DIA) can actually significantly degrade compute performance. Second, whether storing a tensor in a structured sparse tensor format can yield performance improvements also depends on the operation being

performed. For instance, while storing a matrix in DIA instead of CSR can potentially improve the performance of matrix-vector multiplication, it can also potentially degrade the performance of matrix addition with that same DIA matrix and another CSR matrix as inputs by reducing cache spatial locality.

Countless more sparse tensor formats have been developed that are optimized for storing dynamic sparse tensors with sparsity structures that evolve over time. Array-based sparse tensor formats like CSR, DIA, BCSR, and ELL, which store nonzeros using a fixed number of arrays, provide good cache spatial locality and are ideal for storing static sparse tensors with fixed sparsity structures. However, inserting new nonzeros into a tensor that is stored in an array-based sparse tensor format is generally a costly operation, since this may require already-stored nonzeros to be moved around in memory. As Figure 1-3e illustrates, for instance, to insert a new nonzero into a CSR tensor, all subsequent nonzeros must be shifted back so that space can be made for the new nonzero, which incurs significant performance overhead. Formats that are optimized for storing dynamic sparse tensors, by contrast, typically use pointers to link together stored nonzeros, which makes it possible to insert new nonzeros without having to move already-stored nonzeros in memory. Figure 1-3c, for example, shows a pointer-based sparse tensor format that stores nonzeros in each row of a tensor using a binary search tree (BST). Since nodes in a BST do not have to be stored contiguously in memory, a new nonzero can be inserted by simply allocating a new node and attaching it to the rest of the BST without moving any existing node in memory, as Figure 1-3f illustrates. This support for efficient modification, however, comes at the cost of reduced compute performance, since pointer-based data structures like BSTs provide less spatial locality. Thus, to better support applications that have more equal proportions of data modification and compute, many other pointer-based sparse tensor formats instead store nonzeros using blocked data structures like B-trees and block linked lists. Such blocked data structures trade off some modification performance for improved compute performance.

As we just saw, all of these different sparse tensor formats can be ideal in specific circumstances, though none is universally superior. Applications from different domains often have to work with data that benefit from being stored in different formats, and a



(a) Original sparse matrix (b) Original matrix stored in CSR (c) Original matrix stored as BSTs

(d) Modified sparse matrix (e) Modified matrix stored in CSR (f) Modified matrix stored as BSTs

Figure 1-3: Examples of the same matrix stored in CSR and as BSTs. Inserting a new nonzero I into CSR at coordinates $(0, 2)$ requires shifting stored nonzeros in memory and may require reallocating the `crd` and `vals` arrays, whereas inserting the same nonzero into a BST only requires allocating a new node.

single application might also need to use multiple formats to store different data sets or even the same data set for different computations. So to be able to effectively support a wide range of applications, a general-purpose sparse tensor algebra system must be able to efficiently compute on sparse tensors that may be stored in a wide variety of formats. However, the sheer number of different sparse tensor formats makes it challenging to support efficiently computing with all of them. This is because different formats often use vastly dissimilar data structures to store nonzeros, and so computing with sparse tensors that are stored in different formats often require dissimilar code. For example, iterating over a CSR matrix to element-wise multiply it by a dense matrix requires a set of two nested loops that each iterate along a dimension of the CSR matrix (Figure 1-4b). By contrast, performing the same computation with the CSR matrix replaced by a coordinate (COO) format matrix, which explicitly stores the row and column coordinates of every nonzero, only requires a single loop that iterates over both the row and column dimension together (Figure 1-4a). But not only that, as Figure 1-4c shows, efficiently computing on multiple operands that are stored in different sparse tensor formats typically requires code that is more than just a straightforward combination of code that iterate over the operand

```
for (int pB = B1_pos[0];
         pB < B1_pos[1];
         pB++) {
  int i = B1_crd[pB];
  int j = B2_crd[pB];
  int pC = i * N + j;
  int pA = i * N + j;
  A[pA] = B[pB] * C[pC];
}
```

(a) B is COO, C is dense array

```
for (int i = 0; i < M; i++) {
  for (int pB = B2_pos[i];
           pB < B2_pos[i+1];
           pB++) {
    int j = B2_crd[pB];
    int pC = i * N + j;
    int pA = i * N + j;
    A[pA] = B[pB] * C[pC];
  }
}
```

(b) B is CSR, C is dense array

```
int pC1 = C1_pos[0];
while (pC1 < C1_pos[1]) {
  int i = C1_crd[pC1];
  int C1_segend = pC1 + 1;
  while (C1_segend < C1_pos[1] &&
         C1_crd[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_crd[pB2];
    int jC2 = C2_crd[pC2];
    int j = min(jB2, jC2);
    int pA = i * N + j;
    if (jB2 == j && jC2 == j)
      A[pA] = B[pB2] * C[pC2];
    if (jB2 == j) pB2++;
    if (jC2 == j) pC2++;
  }
  pC1 = C1_segend;
}
```

(c) B is CSR, C is COO

Figure 1-4: Examples of code for computing the element-wise multiplication of two matrices that are stored in different combinations of array-based formats. As these examples demonstrate, efficiently computing with sparse tensors can require very dissimilar code depending on what formats are used to store the sparse tensors. The technique I describe in the rest of this dissertation is able to automatically generate all these kernels.

formats individually. And, as Figure 1-5 shows, efficiently computing with tensors that are stored in different pointer-based sparse tensor formats can require even more varied code. For instance, efficiently computing sparse matrix-vector multiplication (SpMV) on a sparse matrix that is stored using BSTs requires code that recursively invokes itself in order to traverse the BSTs and compute on each nonzero (Figure 1-5b). By contrast, performing the same computation on a sparse matrix that is stored using block linked lists only requires a set of two nested loops to iterate over nodes in the linked lists and to iterate over stored nonzeros in each node (Figure 1-5a). And again, as Figure 1-5c shows, efficiently computing on multiple operands stored in different pointer-based sparse tensor formats generally requires code that is very distinct from code that computes with just any one of the operand formats individually.

Such diversity in code that is required to efficiently compute with sparse tensors in disparate formats makes it practically impossible for library developers to manually implement all the kernels that would be needed to fully support a wide range of formats.

```
void map_b(blist* b, double* c, double& a) {
  while (b) {
    for (int32_t p = 0; p < b->B; p++) {
      int32_t j = b->e[p].first;
      a += b->e[p].second * c[j];
    }
    b = b->n;
  }
}

void compute(...) {
  for (int32_t i = 0; i < N; i++) {
    double sum = 0.0;
    map_b(b[i]->h, c, sum);
    a[i] = sum;
  }
}
```

(a) SpMV with matrix stored as block linked lists

```
void map_b(bst* b, double* c, double& a) {
  if (b) {
    if (b->l)
      map_b(b->l, a, c);
    int32_t j = b->e.first;
    a += b->e.second * c[j];
    if (b->r)
      map_b(b->r, a, c);
  }
}

void compute(...) {
  for (int32_t i = 0; i < N; i++) {
    double sum = 0.0;
    map_b(b[i]->r, c, sum);
    a[i] = sum;
  }
}
```

(b) SpMV with matrix stored as BSTs

```
inline uint8_t
iter_bst(uint8_t state, bst*& n,
         call_stack<uint8_t,bst*>& cs,
         int32_t& c, double& v) {
  if (state == 1)
    goto iter_resume1;
  cs.emplace(0, n);
  while (!cs.empty()) {
    n = get<1>(cs.top());
    if (get<0>(cs.top()) == 1)
      goto call_resume1;
    while (n) {
      if (n->l) {
        get<0>(cs.top()) = 1;
        get<1>(cs.top()) = n;
        cs.emplace(0, n->l);
        goto call_end;
call_resume1:;
      }
      c = n->e.first;
      v = n->e.second;
      return 1;
iter_resume1:
      n = n->r;
    }
    cs.pop();
call_end:;
  }
  return 0;
}
```

```
inline uint8_t
iter_blist(uint8_t state, blist*& b,
           int32_t& p, int32_t& c, double& v) {
  if (state == 1)
    goto iter_resume1;
  while (b) {
    for (p = 0; p < b->B; p++) {
      c = b->e[p].first;
      v = b->e[p].second;
      return 1;
iter_resume1:;
    }
    b = b->n;
  }
  return 0;
}

void compute(...) {
  ...
  for (int32_t i = 0; i < N; i++) {
    bst* bn = b[i]->r;
    blist* cn = c[i]->h;
    uint8_t bs = iter_bst(0, bn, bstack, jb, bv);
    uint8_t cs = iter_blist(0, cn, cp, jc, cv);
    while (bs && cs) {
      int32_t j = min(jb, jc);
      if (j == jb && j == jc)
        a[pa++] = bv * cv;
      if (j == jb) bs = iter_bst(bs, ..., bv);
      if (j == jc) cs = iter_blist(cs, ..., cv);
    }
  }
}
```

(c) Element-wise multiplication of a matrix stored as BSTs and a matrix stored as block linked lists

Figure 1-5: Examples of different sparse tensor algebra computations with operands stored in disparate pointer-based formats. As these examples again show, efficiently computing with sparse tensors can require very dissimilar code depending on the format used to store the tensors ((a) vs. (b)) as well as the computation ((a)/(b) vs. (c)). The technique I describe in the rest of this dissertation is also able to automatically generate all of these kernels.

Library developers are then forced to respond by leaving out support for some sparse tensor formats and restricting the set of supported computations. For instance, tensor algebra libraries such as SPLATT [103] and TensorFlow [1], as well as graph processing frameworks such as Aspen [32] and STINGER [34], all support only a handful of formats each. Meanwhile, Intel MKL provides at least partial support for many different formats but, as just one example, only supports computing the addition of CSR or BCSR matrices and does not support directly computing the same operation on matrices in other formats like COO. And even then, since MKL only supports directly computing the addition of two matrices, to add three or more matrices together would actually require decomposing the computation into multiple sub-operations, which significantly reduces performance [53]. Moreover, while MKL supports more formats than many other libraries, ultimately it—and any other hand-optimized library—can only support a fixed set of formats, meaning users with data that do not neatly fit into any of these formats would be left with no good option. Artificial limitations like these render hand-optimized libraries less generally useful, since they can either significantly increase the overhead of performing certain computations that are needed by some applications or simply make it impossible to perform the computations. This motivates a compiler-based technique that can instead automatically generate code to efficiently compute with sparse tensors that may be stored in arbitrary formats, which would eliminate the need for such artificial limitations.

1.3 Our Approach and Contributions

This dissertation shows how to build a compiler that supports efficiently computing arbitrary tensor algebra operations on sparse tensors that may be stored in a wide variety of formats. At a high level, our technique works by representing these different tensor formats as compositions of various *level formats*, each of which uses a distinct data structure to store coordinates of nonzeros along a tensor dimension. To define a level format that uses a fixed set of arrays to store nonzeros, a user simply has to implement an abstract interface we developed that captures how the underlying arrays can be efficiently accessed. Meanwhile, to define a level format that uses a pointer-based data structure to store

nonzeros, a user can simply use a domain-specific language we developed to specify how the data structure organizes stored nonzeros in memory. Additionally, to specify how the results of sparse tensor computations can be stored in a particular level format, a user simply has to implement an abstract interface we developed that captures how the underlying data structure can be efficiently constructed. Then, guided by implementations of these various abstractions for different level formats, the compiler can straightforwardly generate efficient sparse tensor algebra code that are optimized for tensors stored in a wide range of formats. Our abstractions can be implemented for each format independently of all other formats, which makes it practical to add support for a large number of formats. Furthermore, by decoupling specifications of sparse tensor formats from the code generation mechanism, our approach makes it possible to add support for new formats without having to modify the code generator itself. This not only makes the compiler easier to maintain but also makes it possible for users to add support for custom—and potentially novel—formats that are optimized for specialized applications or even data sets.

More concretely, this dissertation makes the following contributions:

- A **levelized abstraction for sparse tensor storage** that models sparse tensor formats as hierarchical compositions of **level formats**, each of which uses a distinct array-based or pointer-based data structure to store a tensor dimension.

- **Coordinate remapping notation**, which generalizes the levelized abstraction for sparse tensor storage to structured sparse tensor formats by casting them as formats for storing tensors with added dimensions.

- An **abstract interface for array-based level formats** that exposes properties about how stored nonzeros are organized in the underlying arrays and that captures how those arrays can be efficiently accessed.

- The **node schema language**, which can be used to define pointer-based level formats by precisely specifying how stored nonzeros are distributed amongst nodes in the underlying data structure and how those nodes are linked together.

- The **attribute query language**, which can be used to precisely specify what statistics need to be known about the result of a sparse tensor computation in order to be able to reserve sufficient memory and coordinate the insertion of nonzeros when constructing the output tensor.

- An **assembly abstract interface** that captures how the results of sparse tensor computations can be efficiently stored in disparate level formats. For array-based formats, this interface captures how the results of attribute queries can actually be used to preallocate memory and coordinate the insertion of nonzeros.

- **Code generation techniques** that, guided by the above abstractions, can emit efficient sparse tensor algebra code to compute on inputs and construct outputs that may be stored in a wide variety of sparse tensor formats.

We have implemented our technique as prototype extensions to the TACO sparse tensor algebra compiler [54]. Our evaluation shows that our technique supports a much wider variety of sparse tensor formats than existing sparse linear and tensor algebra libraries and frameworks. But not only that, sparse tensor algebra code that our technique generates has performance that is competitive with, if not better than, equivalent hand-optimized code that are implemented in the aforementioned libraries and frameworks. Our technique can achieve such levels of performance for a wide range of computations, which enables our technique to, in many cases, significantly outperform hand-optimized libraries and frameworks that have to artificially limit the set of supported operations. As an example, Intel MKL and SPARSKIT [93] only provide hand-optimized code to directly convert sparse matrices between the CSR format and other sparse matrix formats, so converting a matrix from the COO format to the DIA format would require first converting the matrix to a CSR temporary. By contrast, our technique can generate efficient code to directly convert sparse matrices between arbitrary formats, so the same conversion from COO to DIA can be performed 3.96–4.01× faster on average relative to MKL and SPARSKIT. As another example, our technique can generate efficient code to directly compute the sum of a CSR matrix and a dynamic sparse matrix that is stored in a pointer-based format, with the result also stored in CSR. By contrast, such a computation—with the dynamic sparse matrix stored

in any pointer-based format—is simply not readily supported by most sparse linear algebra libraries like MKL or dynamic graph processing frameworks like Terrace [79]. Meanwhile, lower-level libraries such as PAM [109] can be used to implement the operation, though with performance overheads of 6.98× on average relative to our technique due to the need to convert data between different formats.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2 – Sparse Tensor Formats Survey** surveys a selection of sparse tensor formats that have been proposed in the literature and that are supported by existing sparse linear/tensor algebra libraries as well as graph processing frameworks.
- **Chapter 3 – Tensor Storage Decomposition** describes how sparse tensor storage can be modeled as hierarchies of coordinates and how sparse tensor formats can be viewed as compositions of per-dimension level formats that each stores a coordinate hierarchy level, which corresponds to a dimension of a tensor. This chapter also describes how, using coordinate remapping notation, structured sparse tensor formats can be modeled as formats for tensors with added dimensions.
- **Chapter 4 – Supporting Array-Based Formats** presents an abstract interface for array-based level formats and describes how, guided by implementations of this interface, a compiler can generate efficient code to compute on static sparse tensors that are stored in arbitrary array-based sparse tensor formats.
- **Chapter 5 – Supporting Pointer-Based Formats** presents a language for defining pointer-based level formats and describes how, guided by definitions of level formats in this language, a compiler can generate efficient code to compute on dynamic sparse tensors that are stored in arbitrary pointer-based sparse tensor formats.
- **Chapter 6 – Supporting Sparse Tensor Assembly** presents an abstract interface that captures how to efficiently construct sparse tensor data structures and describes

how, guided by implementations of this interface, a compiler can generate code to store the results of sparse tensor computations in arbitrary sparse tensor formats.

- **Chapter 7 – Evaluation** details the results of experiments that demonstrate our technique generates efficient sparse tensor algebra code, as well as shows how different sparse tensor formats are suited to different kinds of operations and data.
- **Chapter 8 – Related Works** compares and contrasts our technique with existing sparse linear/tensor algebra programming systems and compilers, as well as surveys related works on data structure operation synthesis and query language compilation.
- **Chapter 9 – Conclusion and Future Work** summarizes the main takeaways of the dissertation and discusses potential directions for future work.

Chapter 2

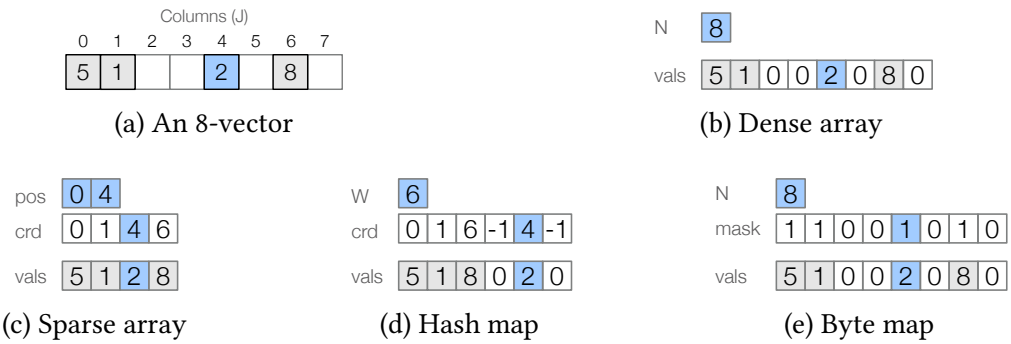
Sparse Tensor Formats Survey

Countless formats for storing sparse tensors have been proposed in the literature. No format is universally superior; each format can be ideal for storing a particular sparse tensor depending on characteristics of the data being stored, the operations being performed, and the available hardware. As a result, many of these formats have been utilized to store data for real-world applications and are supported by commonly-used libraries and frameworks.

This chapter surveys some of these different sparse tensor formats. In particular, Section 2.1 focuses on array-based tensor formats, which use fixed sets of arrays to store nonzeros and are ideal for storing static sparse tensors (i.e., tensors with sparsity structures that remain static during program execution). Meanwhile, Section 2.2 focuses on pointer-based tensor formats, which use pointer-based data structures such as binary trees to store nonzeros and are ideal for storing dynamic sparse tensors (i.e., tensors with sparsity structures that evolve during program execution).

2.1 Array-Based Tensor Formats

Figures 2-1, 2-2, and 2-3 show some examples of tensor formats that use fixed sets of arrays to store elements of tensors. A straightforward way to store any tensor is to simply use a dense array to explicitly store all elements of the tensor, including the zeros. Figure 2-1b shows an example dense storage for a one-dimensional tensor (i.e., a vector). A desirable feature of dense arrays is that the value at any coordinate can be accessed in constant



(a) An 8-vector (b) Dense array

(c) Sparse array (d) Hash map (e) Byte map

Figure 2-1: Examples of the same vector stored in different array-based formats. The array elements that are shaded blue all encode the same nonzero.

time. Storing a sparse tensor in a dense array, however, is inefficient as a lot of memory is wasted to store zeros. Furthermore, performance is lost computing with these zeros even though they do not meaningfully contribute to the result. For tensors that possess many large dimensions, it may even be impossible to use dense arrays due to lack of available memory. A data set of Amazon product reviews [71], for instance, can be represented by a 4.8 million $\times$ 1.8 million $\times$ 1.8 million tensor, which would require 107 exabytes of storage if elements are stored as double-precision floats in a dense array.

The simplest way to efficiently store a sparse tensor is as a list of its nonzeros, with the full coordinates and value of every nonzero being explicitly stored (Figures 2-1c, 2-2b, and 2-3b). This is typically known as the coordinate (COO) format [10]. In contrast to dense arrays, COO tensors consume only $\Theta(\text{nnz})$ memory, so using the COO format instead of dense arrays to store sparse tensors can reduce memory usage by orders of magnitude. In addition, many common file formats for storing tensors, such as the Matrix Market exchange format [75] and the FROSTT sparse tensor format [100], closely mirror the COO format. This minimizes the cost of loading a tensor from file since each nonzero can be inserted into a COO tensor by simply appending its coordinates and value to the `crd` and `vals` arrays, as long as those arrays have already been pre-allocated with sufficient space.

Unlike dense arrays though, the COO format does not provide efficient random access, which can limit the performance of multiplicative operations. Hash maps eliminate this drawback by storing tensor coordinates in a randomly accessible hash table (Figure 2-1d). However, hash maps do not support efficiently iterating over stored nonzeros in order,

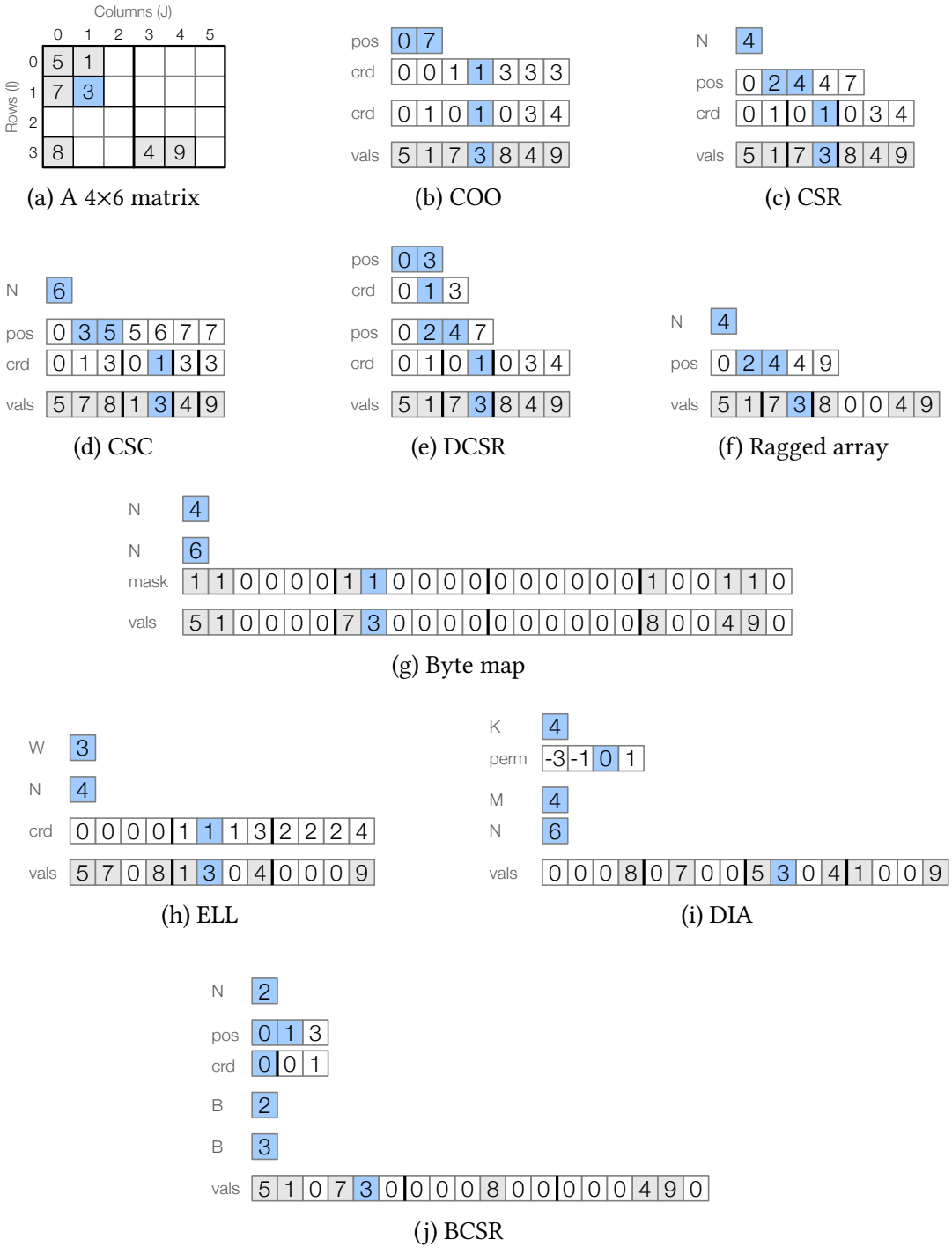
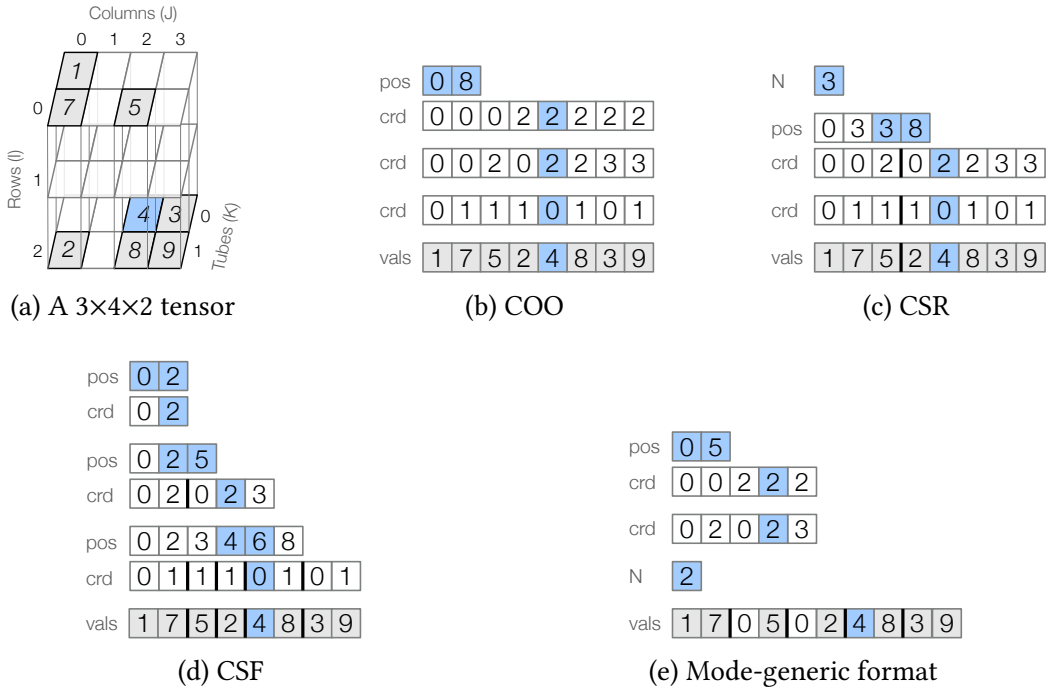


Figure 2-2: Examples of the same matrix stored in different array-based formats. The array elements that are shaded blue all encode the same nonzero.



(a) A 3×4×2 tensor (b) COO (c) CSR

(d) CSF (e) Mode-generic format

Figure 2-3: Examples of the same three-dimensional tensor stored in different array-based formats. The array elements that are shaded blue all encode the same nonzero.

which in turn limits the performance of additive operations. Alternatively, dense arrays can be extended with a byte map that records whether each stored value is non-zero (Figure 2-1e). Such a byte map can effectively guard accesses into the dense array by only loading an element from the dense array if the byte map indicates that it is non-zero. If the stored tensor is sufficiently sparse, then this can reduce total memory traffic at run-time when computing on the tensor, which can increase performance as a result.

Another drawback of the COO format is that it redundantly stores row coordinates. In Figure 2-2b, for instance, the row coordinates of the last three nonzeros are all explicitly stored even though those nonzeros belong in the same row of the matrix. The compressed sparse row (CSR) format for sparse matrices (Figure 2-2c) compresses out such redundant row coordinates by using an auxiliary array (pos in Figure 2-2c) to keep track of which nonzeros belong to each row of a matrix. This can improve the runtime performance of computations that are typically memory bandwidth-bound, such as SpMV. The compressed sparse column (CSC) format (Figure 2-2d) follows the same basic principle but instead compresses out redundant column coordinates. The doubly compressed sparse row (DCSR)

format [20] achieves additional compression for hypersparse matrices by only storing the rows that contain nonzeros (Figure 2-2e). The CSR format can also generalized to support storing n-dimensional tensors by using an additional `crd` array to store each additional dimension (Figure 2-3c), effectively storing each $(n-1)$-dimensional subtensor in COO. Smith and Karypis [102] further describe another generalization of the CSR format, called compressed sparse fiber (CSF), that fully compresses every dimension by using additional `pos` arrays to avoid storing redundant coordinates (Figure 2-3d). Sparse tensors stored in any of these compressed formats, however, are particularly costly to assemble or modify, since inserting any new nonzero can require all the other already-stored nonzeros to have to be moved around in memory.

Many applications work with tensors that have non-zero elements distributed in regular patterns. For instance, a matrix that encodes vertex-edge connectivity of a well-formed unstructured mesh typically contain similar numbers of nonzeros in all rows of the matrix. The ELLPACK (ELL) format [52] exploits this by storing the same number of elements for every row of a matrix (Figure 2-2h). This enables the ELL format to only store the column coordinates and values of nonzeros, with the row coordinate of each nonzero being implicitly encoded by its position in the `crd` and `vals` arrays. Additionally, the ELL format stores nonzeros that belong to adjacent rows of a matrix (but that share the same relative position within their respective rows) contiguously in the `crd` and `vals` arrays, and this enables certain computations like SpMV to be efficiently vectorized [31]. If nonzeros in a matrix are further known to be clustered within a few densely-filled diagonals, as is generally the case with matrices that encode stencils applied to regular grids for instance, then the diagonal (DIA) format [94] can exploit this by only storing the offsets of those diagonals from the matrix's main diagonal (Figure 2-2i). This enables the DIA format to forgo explicitly storing the column coordinates of nonzeros altogether (since they can be simply computed from the offsets of the diagonals), further reducing memory usage. On the other hand, if nonzeros in a matrix are clustered along the start of each row, then ragged arrays [110] can exploit this by storing all elements (including zeros) up to the last nonzero in each row (Figure 2-2f). This again makes it possible to forgo storing column coordinates, since the column coordinate of any nonzero is simply its offset from the start

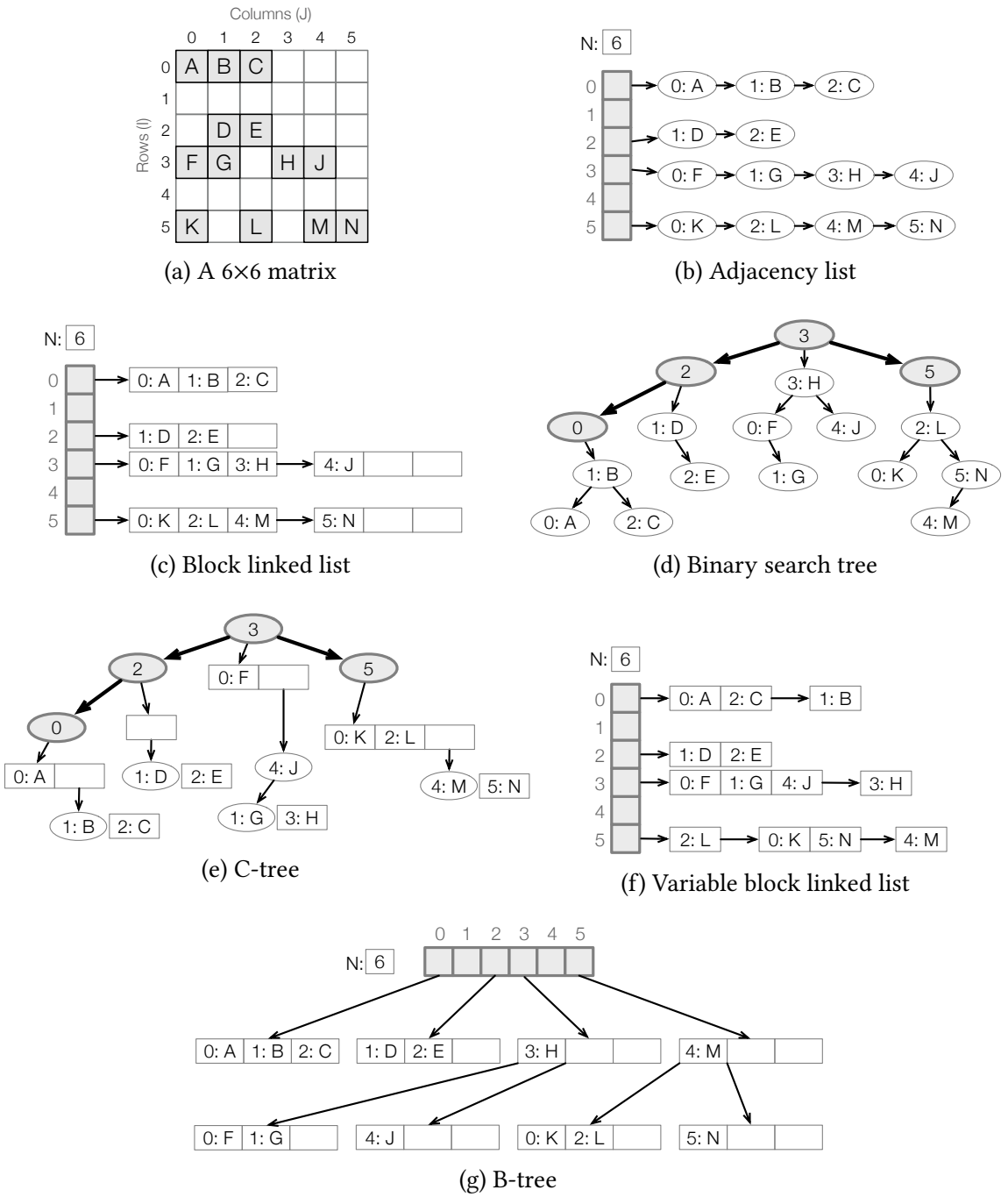
of its containing row's segment in the ragged array. However, for matrices that do not exhibit specific types of sparsity structure, storing them in formats like DIA and ELL may actually increase memory usage and degrade compute performance. The DIA format, for instance, cannot store only parts of a diagonal. As a result, using DIA to store an unstructured sparse matrix that has nonzeros distributed amongst many sparsely-filled diagonals would require all the zero elements within those diagonals to be explicitly stored as well. Computing on the matrix would then require redundantly computing with many zero elements, which degrades performance.

The block compressed sparse row (BCSR) format [43], meanwhile, generalizes CSR by replacing each nonzero in the `vals` array with a dense block of nonzeros instead (Figure 2-2j). This effectively amortizes the overhead of storing the coordinates of nonzeros as well as exposes opportunities for vectorization, which makes BCSR ideal for storing inherently blocked matrices such as those arising from finite element analysis. The mode-generic sparse tensor format, proposed by Baskaran et al. [12], generalizes the idea of BCSR to higher-dimensional tensors (Figure 2-3e). It stores a tensor as a sparse collection of dense blocks that may have any number of dimensions, with the coordinates of the blocks effectively stored in the COO format (i.e., the `crd` arrays).

2.2 Pointer-Based Tensor Formats

The formats described in the previous section are ideal for storing static sparse tensors, since the use of arrays to compactly store nonzeros ensures good cache spatial locality and improves compute performance as a result. For the same reason though, array-based sparse tensor formats are generally not well-suited for storing dynamic sparse tensors, since modifying them (e.g., by inserting nonzeros) requires already-stored nonzeros to be moved around in memory and incurs significant overhead.[1] To address this, many different pointer-based sparse tensor formats have been developed, each with distinct trade-offs.

[1]Dense arrays do support efficient in-place modification by effectively pre-allocating storage for all potential new nonzeros. As mentioned in the previous section though, they cannot be used to store large data sets that are common in many real-world applications.



(a) A 6×6 matrix

(b) Adjacency list

(c) Block linked list

(d) Binary search tree

(e) C-tree

(f) Variable block linked list

(g) B-tree

Figure 2-4: Examples of the same matrix stored in different pointer-based formats.

Figure 2-4 shows several representative examples of pointer-based formats for storing dynamic sparse matrices (or adjacency matrices that represent dynamic graphs). A standard way of storing a dynamic sparse matrix is as a collection of adjacency lists, each of which stores the nonzeros in a single row of a matrix. Each adjacency list can be stored as a

linked list [28], with each node in the linked list storing the column coordinate and value of one nonzero (Figure 2-4b). This representation enables new nonzeros to be efficiently added to a matrix by simply appending them to the appropriate adjacency lists, which can be done without moving any existing nonzero in memory. Additionally, the collection of adjacency lists may itself be stored as a linked list, forming the list of lists representation; this enables new rows to be efficiently added to a matrix as well.

One drawback with linked lists though is that, when iterating over stored nonzeros, each access can potentially incur a cache miss since nodes in a linked list are typically not stored contiguously in memory. This increases the overhead of accessing nonzeros and thereby reduces performance when computing with tensors that are stored as linked lists. To address this, some high-performance graph processing frameworks like STINGER [34] instead use block linked lists to store multiple nonzeros in each node (Figure 2-4c), effectively amortizing the overhead of each node access. In a typical block linked list, every node contains an array of the same size and is able to store the same maximum number of nonzeros. However, some frameworks like GraphOne [61] use variable block linked lists that allow different nodes to contain arrays of different sizes, thus enabling nodes to store different maximum numbers of nonzeros (Figure 2-4f). This allows updates to be efficiently batched, with each batch of new nonzeros inserted as just a single new node.

Another way of representing a dynamic sparse matrix is to use (balanced) binary search trees (BSTs) to store the set of nonzeros within each row as well as the set of non-empty rows (Figure 2-4d) [32]. Using BSTs to store nonzeros enable new nonzeros to be efficiently inserted while also keeping the data structure sorted, which is useful for computations that must access nonzeros in order. Again though, to amortize the overhead of accessing nodes in a tree, many high-performance graph processing frameworks instead use block tree data structures that store multiple nonzeros in each node. For instance, Aspen [32] represents each row of a dynamic graph's adjacency matrix using a C-tree, which stores only a subset of nonzeros (i.e., *heads*) directly in a BST (Figure 2-4e). The remaining nonzeros, meanwhile, are stored in either a *prefix* (which contains all nonzeros that have smaller coordinates than any head element) or in chunks (i.e., *tails*) that are each associated with a distinct head element. Similarly, Terrace [79] supports storing each row of an adjacency

matrix using B-trees, which generalize BSTs in a different way by allowing each node to store more than two children in addition to storing multiple nonzeros (Figure 2-4g).

2.3 Chapter Summary

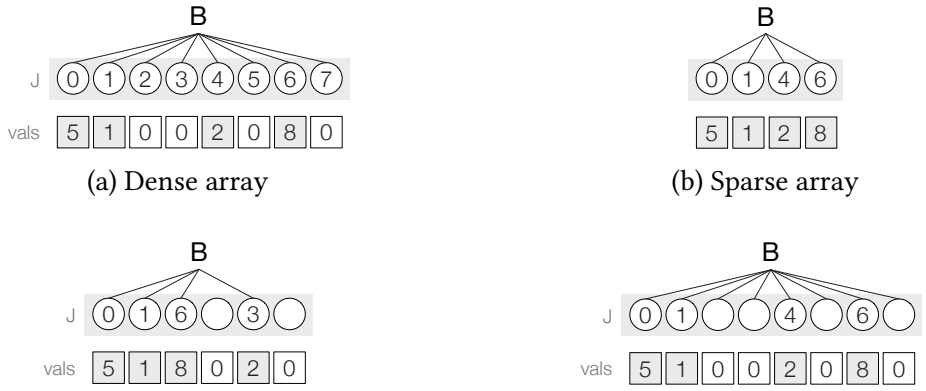
In this chapter, we took a closer look at some of the many sparse tensor formats that have been proposed in the literature and that are supported by widely-used libraries and frameworks. We also saw that all of these formats possess distinct advantages and disadvantages, and so to effectively support a wide range of real-world applications, a general-purpose system must be able to efficiently compute with tensors that are stored a wide range of formats. In the next four chapters, we will see how our technique can abstractly represent disparate formats—including all the ones we saw in this chapter—and we will see how this lets our technique generate code that efficiently compute on sparse tensors stored in these formats.

Chapter 3

Tensor Storage Decomposition

As we saw in Section 1.2, to be able to efficiently compute with sparse tensors that may be stored in arbitrary combinations of formats, a compiler must be able to generate specialized code for any combination of input (and output) tensor formats. Since the number of possible combinations of distinct tensor formats is exponential in the number of supported formats though, it is infeasible for a compiler to exhaustively hard-code support for every format.

In the rest of this dissertation, I show how a compiler can represent disparate tensor formats using a common set of abstractions, which lets the compiler reason about how to efficiently work with tensors in different formats without having to hard-code for any particular format. This chapter, in particular, shows how a wide range of sparse tensor formats, including all those described in Chapter 2, can be expressed as compositions of per-dimension formats. The same per-dimension formats can be straightforwardly recombined in many different ways to express distinct tensor formats, which is one way by which our technique can generalize to a wide variety of tensor formats. And, as we will see later, decomposing tensor formats into per-dimension formats lets a compiler generate sparse tensor algebra code in a dimension-by-dimension fashion, which enables our technique to support tensors that have arbitrary numbers of dimensions.



(a) Dense array

(b) Sparse array

(c) Hash map

(d) Byte map

Figure 3-1: Coordinate hierarchy representations of the same vector (shown in Figure 2-1a) stored in different formats. A coordinate hierarchy's structure reflects how the underlying storage format encodes a tensor's nonzeros.

3.1 Coordinate Hierarchies

The idea of per-dimension formats can be understood by viewing tensor storage as a hierarchy of coordinates, where each level in the *coordinate hierarchy* encodes the coordinates of nonzeros along a particular tensor dimension. Figures 3-1 and 3-2 show example of coordinate hierarchies that represent tensors stored in distinct formats. Each path from the root to a leaf in a coordinate hierarchy encodes the full coordinates of a tensor element, with the corresponding value of the tensor element shown below. In Figure 3-2b, for instance, the rightmost path represents the tensor element $B(3, 4)$, which has a value of 9.

A coordinate hierarchy has one level (shaded light gray in Figure 3-2) for each dimension of a tensor. Each position (corresponding to a node) in a level may encode some coordinate (the label in the node) along the tensor dimension that the level represents. Alternatively, a position may contain an unlabeled node, which encodes no coordinate and reflects padding in the underlying physical storage. The unlabeled nodes in Figure 3-2e, for instance, represent parts of each diagonal that exceed the bounds of the matrix, and these unlabeled nodes correspond to padding in the matrix's DIA representation (Figure 2-2i). Each node in a level may also be connected to a parent node 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 parent row coordinate 3. Meanwhile,

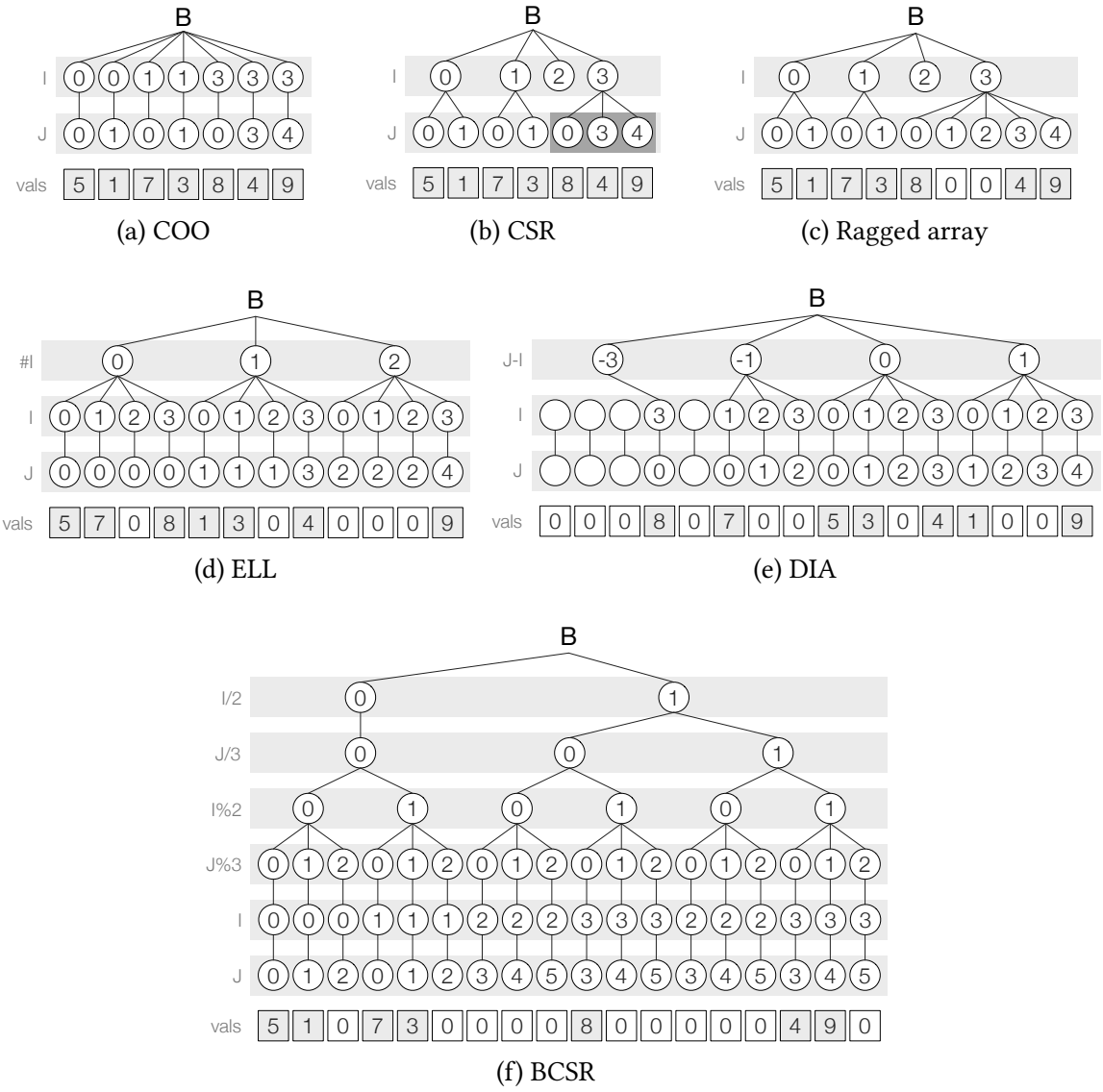


Figure 3-2: Coordinate hierarchy representations of the same matrix (shown in Figure 2-2a) stored in different formats.

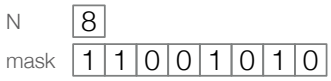
a coordinate's *ancestors* refer to the set of coordinates that are encoded along the path from its parent to the root.

By representing tensor storage in this manner, we can then view any tensor format as a composition of *level formats* that each stores a single level of a coordinate hierarchy (including all nodes in the level along with edges that connect them to their parents). Different level formats can store coordinate hierarchy levels in very different ways; some level formats may implicitly encode coordinates (e.g., as intervals between store bounds), while others might instead explicitly store coordinates (e.g., in segmented vectors or

binary search trees). Below I show examples of level formats that can be composed to express all the tensor formats that were described in Section 2.1. In particular, given a parent coordinate in the $(i-1)$-th level of a coordinate hierarchy (and potentially also a grandparent coordinate in the $(i-2)$-th level as well as other ancestor coordinates), these level formats encode its child coordinates in the i-th level as follows:

Dense levels store the size of the corresponding tensor dimension (N) and encode coordinates in the interval $[0, N)$. The value of N is part of the shape of the tensor and is known statically. Figure 3-2b shows the row (I) dimension of a CSR matrix encoded as a `dense` level using a scalar variable as shown below.

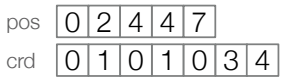
Masked levels similarly encode coordinates in the interval $[0, N)$ (where N represents the size of the corresponding tensor dimension), but potentially with holes in this interval (corresponding to zero elements) stored in the Boolean `mask` array. Figure 3-1d shows the column (J) dimension of a byte map row vector encoded as a `masked` level in the scalar variable and array shown below.



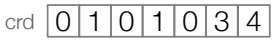
Sliced levels similarly store an upper bound W and encode coordinates in the interval $[0, W)$. Unlike with dense levels though, the value of W is not known statically and, at run-time, can assume any value between 0 and the size of the corresponding tensor dimension. Figure 3-2d shows the relative positions of nonzeros within rows of an ELL matrix (represented by the $\#I$ dimension) encoded as a `sliced` level using a scalar variable as shown below.

Compressed levels store coordinates in a segment of the `crd` array, with the segment bounds stored in the `pos` array. Figure 3-2b shows the column dimension of a CSR

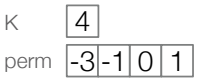
matrix encoded as a `compressed` level in the arrays shown below. Given a parent coordinate 1, for instance, the level encodes two child coordinates 0 and 1, stored in the `crd` array between positions `pos[1]` = 2 (inclusive) and `pos[2]` = 4 (exclusive).



Singleton levels store a single coordinate with no sibling in the `crd` array. Figure 3-2a shows the column dimension of a COO matrix encoded as a `singleton` level in the array shown below.



Squeezed levels store K coordinates in the `perm` array. Unlike `compressed` levels, which can encode distinct child coordinates for different parent coordinates, `squeezed` levels encode the same child coordinates for all parent coordinates. Figure 3-2e shows the $J-I$ dimension representing non-empty diagonals of a DIA matrix encoded as a `squeezed` level in the scalar variable and array shown below.

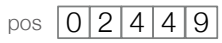


Range levels encode the coordinates in an interval with bounds being functions of the parent coordinate and dimension sizes N and M. Figure 3-2e shows the row dimension of a DIA matrix encoded as a `range` level using the scalar variables shown below. Given a parent coordinate 1, for instance, the level encodes coordinates between $\max(0, -1) = 0$ (inclusive) and $\min(4, 6 - 1) = 4$ (exclusive).

Offset levels encode a single coordinate with no sibling that is computed as the sum of the parent coordinate and the grandparent coordinate. Figure 3-2e shows the column dimension of a DIA matrix encoded as an `offset` level. Given a parent coordinate 3 and a grandparent coordinate 1, for instance, the level encodes the coordinate 3 + 1 = 4.

Block levels encode a single coordinate with no sibling that is computed as the sum of the grandparent coordinate and the product of the coordinate in the $(i-4)$-th level and a block size B. Figure 3-2f shows the column dimension of a BCSR matrix encoded as a `block` level using the scalar variable shown below. Given a grandparent coordinate 2 and coordinate 1 in the $(i-4)$-th level with block size 3, for instance, the level encodes the coordinate $2 + 1 \times 3 = 5$.

Ragged levels encode coordinates in the interval $[0, S)$, which are stored contiguously in a segment of the level. The value of S for each parent coordinate at position p is encoded as the difference between `pos[p]` and `pos[p+1]`, which store the start and end of the segment that correspond to the parent coordinate's children. Figure 3-2c shows the column dimension of a ragged array matrix encoded as a `ragged` level in the array shown below. Given a parent coordinate 1, for instance, the level encodes coordinates between 0 (inclusive) and `pos[2]` − `pos[1]` = 2 (exclusive).



Hashed levels store coordinates in a segment of length W of a hash table (`crd`). Figure 3-1c shows the column dimension of a hash map row vector encoded as a `hashed` level using the scalar variable and array shown below, with −1 marking empty buckets.

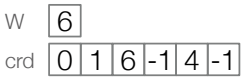
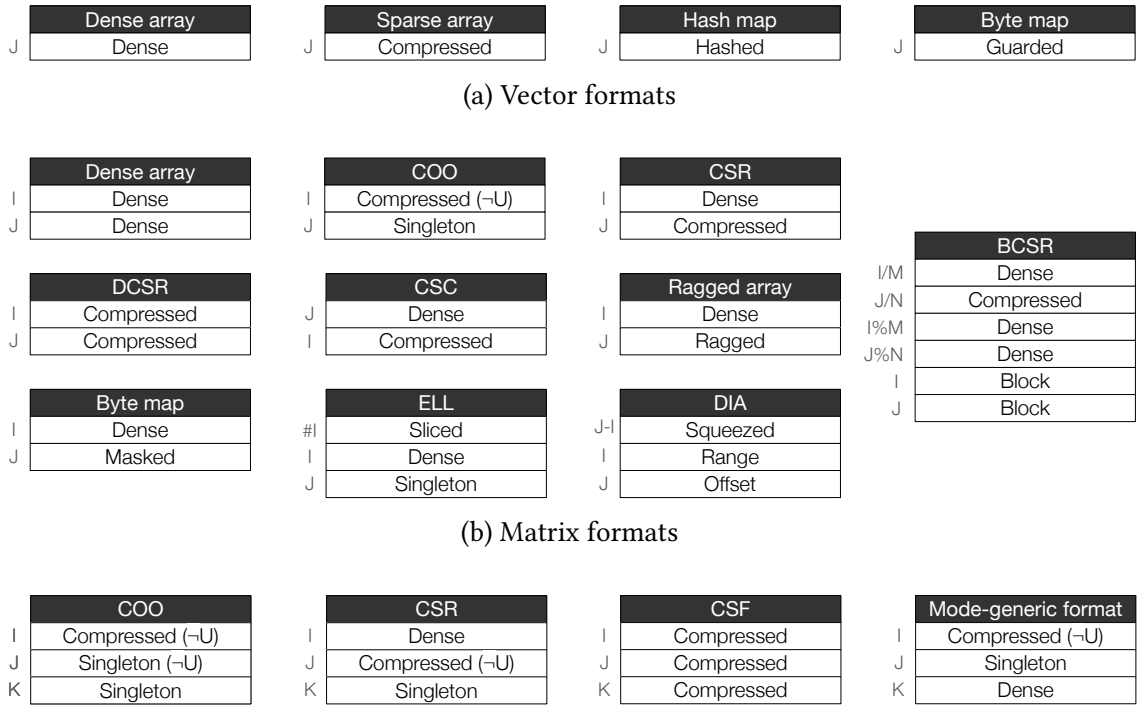


Figure 3-3 shows how these level formats can be composed in different ways to express all the tensor formats that were described in Section 2.1. The same level formats, however, can also be combined in other ways to express additional sparse tensor formats. For instance, the composition (`compressed`, `dense`) expresses a new sparse matrix format that efficiently stores matrices with mostly empty rows and a few additional rows that are densely filled. Furthermore, all these array-based level formats are formally defined by implementing a



(a) Vector formats

(b) Matrix formats

(c) Three-dimensional tensor formats

Figure 3-3: Common tensor formats expressed as compositions of level formats. As described in more depth in Section 3.2, we cast structured sparse matrix formats such as DIA as formats for higher-dimensional remapped tensors. The label beside a level format identifies the tensor dimension it represents. Unless otherwise stated, all level formats other than `hashed` are ordered and unique (described in more depth in Section 4.1.2); `hashed` is unordered and unique. (¬O) denotes an unordered variant of a level format, and (¬U) denotes a non-unique variant of a level format.

shared set of abstract interfaces, and users can define new level formats on top of the ones described above by simply implementing these same interfaces; Chapters 4 and 6 describe these interfaces in depth. In these ways, the coordinate hierarchy abstraction is able to support a wide variety of disparate sparse tensor formats.

The coordinate hierarchy abstraction assumes by default that different levels are stored using separate data structures in memory, though the abstraction also permits multiple levels to be stored using the same shared data structures in memory. As an example, each `block` level and its `dense` parent level in a BCSR matrix's coordinate hierarchy representation have to store the same block dimension size, and so these levels can be configured to share a single variable that contains the block dimension size. Allowing multiple levels to share data structures in memory also enables array-of-structs (AoS) COO to be expressed

as a composition of level formats by defining variants of `compressed` and `singleton` that use a shared array to store coordinates along multiple dimensions in an interleaved fashion.

As we will see in the next three chapters, the coordinate hierarchy abstraction simplifies sparse tensor algebra code generation in several ways. First, decomposing tensor formats into per-dimension formats lets a compiler generate code in a dimension-by-dimension fashion, which enables our technique to support tensors that have arbitrary numbers of dimensions. Second, the structure of a coordinate hierarchy reflects how nonzeros are physically encoded in memory, and so a compiler can generate optimized code for specific operand formats by only reasoning about possible structures of coordinate hierarchies that represent tensors stored in those formats. As an example, the coordinate hierarchy representation of a COO matrix (Figure 3-2a) consists of distinct chains of coordinates that each encodes a single nonzero, reflecting the fact that COO stores the full coordinates of every nonzero. If a compiler knows an input tensor is stored in a format like COO that enforces such a chain structure, then the compiler can generate code that optimizes iteration over the tensor by omitting explicit loops over the tensor's inner dimensions (as in Figure 1-4a). By contrast, the coordinate hierarchy representation of a CSR matrix (Figure 3-2b) is tree-structured, and elements that belong in the same row of the matrix share a row coordinate parent. This tree structure reflects the fact that the CSR format compresses out redundant row coordinates (using the `pos` array) and prevents the aforementioned optimization from being applied. Section 4.1.2 shows how a shared abstract interface for level formats can statically expose such variations in the structure of a matrix's coordinate hierarchy representation to a compiler.

3.2 Coordinate Remapping

Level formats typically group tensor elements that share the same ancestor coordinates (i.e., that belong in the same subtensor) together in memory. As a result, compositions of level formats naturally store tensor elements in lexicographic order of their coordinates. However, structured sparse tensor formats such as DIA and BCSR typically do not store tensor elements lexicographically by their (original) coordinates; DIA, for instance, groups

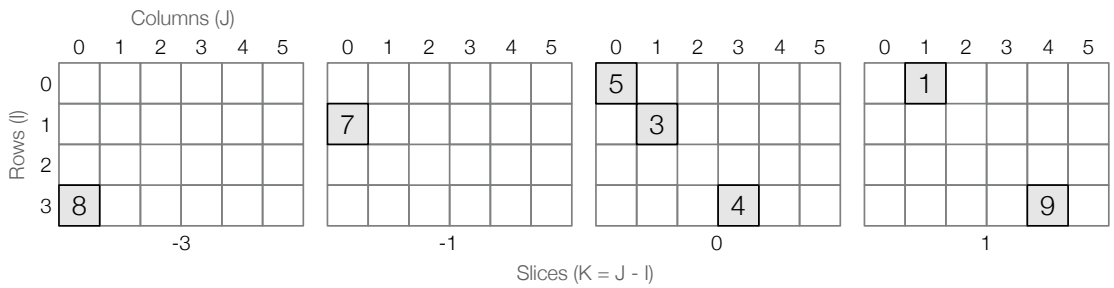


Figure 3-4: The matrix in Figure 2-2a can be transformed to a three-dimensional tensor where each slice contains all nonzeros that lie on the same diagonal in the original matrix. The lexicographic coordinate ordering of nonzeros in the resulting tensor matches the order in which nonzeros are stored in DIA (Figure 2-2i). Such a transformation can be formalized by the coordinate remapping `(i, j) -> (j-i, i, j)`.

together elements that belong in the same diagonal, while BCSR groups together elements that belong in the same block.

We can nevertheless express any structured sparse tensor format as a composition of level formats by casting it as a format for storing tensors with added dimensions. For example, a DIA matrix can be cast (or *remapped*) as a three-dimensional tensor where each slice contains only nonzeros that lie on the same diagonal, as shown in Figure 3-4. Comparing this remapped tensor to the original DIA matrix (Figure 2-2i), we can observe that the lexicographic coordinate ordering of nonzeros in the remapped tensor accurately reflects how the DIA format groups together nonzeros. This lets us decompose DIA into three level formats: one (`squeezed`) that stores the set of nonempty diagonals (or slices of the remapped tensor) in a `perm` array of size `K`, another (`range`) that encodes the set of rows in each diagonal (or slice of the remapped tensor), and a third (`offset`) that encodes the column coordinates of nonzeros.

We formalize such remappings by developing a new language that we call *coordinate remapping notation*. Figure 3-5 shows the syntax of coordinate remapping notation. Statements in coordinate remapping notation specify how elements in an input tensor map to elements in an output tensor, which can potentially have additional dimensions. For instance, given a matrix A as input, the statement

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

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

Figure 3-5: Syntax of coordinate remapping notation.

maps every element A_{ij} to the corresponding element in the $(j - i)$-th slice of the three-dimensional remapped tensor. Applying this coordinate remapping to any matrix transforms it to a three-dimensional tensor where each slice contains all nonzeros that lie on the same diagonal in the original matrix. So rather than regarding DIA as a format for storing matrices, we can instead interpret DIA as a format for storing three-dimensional tensors obtained by applying the above coordinate remapping to the matrices.

Similarly, the BCSR format partitions a matrix into fixed-sized $M \times N$ blocks and stores elements of each block contiguously in memory [43]. Such grouping of nonzeros can be obtained with the coordinate remapping

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

which assigns components that lie within the same block to the same subtensor (identified by coordinates $(i/M,\ j/N)$) in the remapped tensor. We can then interpret BCSR as a format for storing six-dimensional tensors obtained by applying the above remapping to matrices. This, in turn, lets us express BCSR as a composition of six level formats as shown in Figure 3-3b. And by using the `dense` level format in particular to encode coordinates within blocks (i.e., the dimensions indexed by `i%M` and `j%N`), our specification of BCSR exposes the fact that each stored block can be expected to be densely filled.

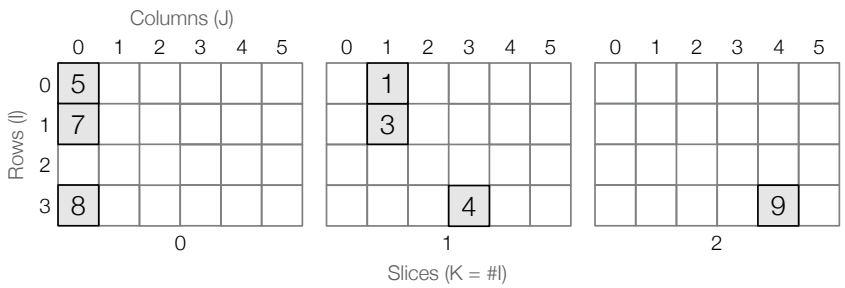


Figure 3-6: Result of applying the coordinate remapping `(i, j) -> (#i, i, j)` to the matrix in Figure 2-2a, assuming nonzeros are iterated over during remapping in the same order as they are stored in Figure 2-2c.

Coordinate remapping notation can express complex orderings of nonzeros. The coordinate remapping below, for instance, groups together nonzeros that lie within the same constant-sized $N \times N \times N$ block and also orders the blocks as well as the nonzeros within each block in Morton order [73]:

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

This remapping exactly captures how the HiCOO tensor format orders stored nonzeros in memory [66]. In particular, nested let expressions are used to first define variables `r`, `s`, and `t` as the coordinates of each block and variables `u`, `v`, and `w` as the coordinates of each nonzero within a block. The remapping then computes the Morton code of each block and each nonzero within a block by interleaving the bits of those previously defined coordinates using bitwise operations.

Coordinate remapping notation also provides counters, which are denoted by the '`#`' operator. Counters map nonzeros that share the same specified coordinates to distinct slices in the remapped tensor. For instance, as Figure 3-6 illustrates, the coordinate remapping

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

can assign the k-th nonzero in each row of a matrix to the k-th slice in the remapped tensor, ensuring nonzeros with the same i coordinate are remapped to distinct slices. This remapping effectively groups together up to one nonzero from each row of the original matrix, accurately reflecting how formats like ELL and JAD [92] store nonzeros in memory.

Finally, coordinate remapping can also be used to express non-row major orderings of nonzeros. For instance, the coordinate remapping

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

groups together nonzeros in a matrix by column, capturing how column-major formats such as CSC store nonzeros in memory.

3.3 Pointer-Based Formats

So far, we have implicitly assumed that all nodes in a coordinate hierarchy level must share the same logical address space, with each node being identified by a unique position in the level. This accurately reflects how array-based level formats store (coordinates of) nonzeros using fixed sets of arrays whose elements can be addressed by their positions. However, the assumption does not accurately model level formats that store nonzeros using pointer-based data structures, which store elements non-contiguously and typically at arbitrary locations in memory. To be able to represent such pointer-based level formats, we generalize the coordinate hierarchy abstraction so that nodes in a level—along with their children—can exist in distinct address spaces. So, for instance, the coordinate hierarchy representation of a matrix stored using binary search trees would have multiple address spaces that each contains a distinct nonzero. Nonzeros in different address spaces can share the same position within their respective address spaces, which reflects the fact that the nonzeros are stored separately (i.e., in different nodes of a binary search tree). The address spaces themselves, though, do not have identifying positions, which reflects the fact that nodes of a binary search tree can be allocated anywhere in memory (i.e., there is no constraint on how nodes must be placed relative to each other in memory).

We can then define additional pointer-based level formats like `bst`, `ctree`, and `blist` that use pointer-based data structures such as binary search trees, C-trees, and block linked lists, respectively, to store coordinate hierarchy levels with multiple address spaces. This enables us to, for instance, express Aspen's adjacency matrix representation (Figure 2-4e) as the composition (`bst`, `ctree`), indicating that the set of nonempty rows are stored using a binary search tree while the set of nonzero columns for each row are stored using a C-tree.

A tensor format may also be composed of both array-based level formats and pointer-based level formats. For example, the composition (`dense`, `blist`) describes a tensor format that stores a matrix as a dense array of block linked lists, each of which stores a row of the matrix (Figure 2-4c); this format is akin to STINGER's adjacency matrix representation. Chapter 5 describes how all the aforementioned pointer-based level formats—and many more—can be precisely defined using a shared language.

3.4 Supporting Non-Zero Fill Values

While we have thus far assumed that sparse tensor storage always compresses out zero elements, many real-world applications actually work with tensors that mostly contain elements of another value. For example, consider a tensor that encodes distances between directly-connected points in a road network, with two points having a distance of ∞ if they are not directly connected by a road. Most elements in such a tensor would likely be ∞ instead of zero, since most pairs of points in the road network are unlikely to be directly connected. To support applications like these, we can generalize the idea of sparsity so that any tensor is sparse as long as most of its elements have the same (possibly non-zero) value, which we refer to as the tensor's *fill value*. As it turns out, such a sparse tensor can still be stored using any of the formats we have seen previously by simply assuming that compressed-out elements contain the tensor's fill value. In a similar way, the coordinate hierarchy abstraction can represent sparse tensor formats that compress out non-zero fill values, with any element simply assumed to contain the tensor's fill value if it is not explicitly encoded by a path in the coordinate hierarchy. This generalization is explored in more depth in the work by Henry et al. [42].

The generalized notion of sparsity described above can be further extended by allowing different subtensors within the same tensor to have different fill values (i.e., *dynamic fill values*). As it turns out, this enables tensor formats that compress out repeated elements using lossless compression schemes (such as run-length encoding) to be expressed in terms of level formats that encode dynamic fill values. This generalization is explored in more depth in the work by Donenfeld et al. [33].

3.5 Chapter Summary

In this chapter, we saw how a wide range of sparse tensor formats can be expressed as compositions of level formats that store coordinates of nonzeros using distinct array-based or pointer-based data structures. In the next two chapters, we will see how array-based and pointer-based level formats can be precisely defined, and we will see how specifications of level formats can be used to generate efficient code that compute on tensors stored using the level formats. Furthermore, in Chapter 6, we will see how our technique can generate efficient code for storing the results of sparse tensor operations in disparate level formats.

Chapter 4

Supporting Array-Based Formats

As we have just seen, a wide range of disparate sparse tensor formats can all be expressed as compositions of per-dimension level formats. In this chapter, I first show how array-based level formats, which use fixed sets of arrays to store coordinates of nonzeros, can be precisely defined by implementing a common abstract interface that exposes properties of the stored data and that captures how the stored data can be efficiently accessed (Section 4.1). I then describe how, guided only by implementations of this abstract interface, a compiler can generate efficient code to compute tensor algebra operations on operands that may be stored in arbitrary combinations of array-based sparse tensor formats (Section 4.2). Our abstract interface can be implemented for each format completely independently of other formats, which makes it practical to add support for a large number of formats. And by utilizing an abstract interface that decouples specifications of sparse tensor formats from the code generation mechanism, our approach makes it possible to add support for new formats without requiring modifications to the compiler itself. The result is a compiler that can support a wide range of formats while still being easy to use and maintain.

4.1 Abstract Interface for Array-Based Level Formats

Our abstract interface for array-based level formats exposes *capabilities* and *properties* of coordinate hierarchy levels that are stored using different level formats. Capabilities of level formats, which are explained in more depth in Section 4.1.1, capture how stored coordinates

Table 4.1: Supported capabilities and properties of each level format. V and P indicate that a level format supports coordinate value iteration and coordinate position iteration, respectively. (✓) indicates that a level format can optionally possess or not possess a particular property.

Level Format	Capabilities		Properties					
	Iteration	Locate	Full	Ordered	Unique	Zeroless	Branchless	Compact
`dense`	V	✓	✓	(✓)	(✓)	(✓)		✓
`masked`	V	✓		(✓)	(✓)	(✓)		
`sliced`	V			(✓)	(✓)	(✓)		✓
`compressed`	P		(✓)	(✓)	(✓)	(✓)		✓
`singleton`	P		(✓)	(✓)	(✓)	(✓)	✓	✓
`squeezed`	P			(✓)	(✓)	(✓)		✓
`range`	V	✓		(✓)	(✓)	(✓)		
`offset`	P			(✓)	(✓)	(✓)	✓	✓
`block`	P			(✓)	(✓)	(✓)	✓	✓
`ragged`	V	✓		(✓)	(✓)	(✓)		✓
`hashed`	P	✓			(✓)	(✓)		

of tensor elements can be efficiently accessed by iterating over or randomly accessing the underlying arrays. Meanwhile, properties of level formats, which are explained in more depth in Section 4.1.2, expose relevant characteristics of the data structures used to store coordinates of tensor elements. Table 4.1 identifies the properties and supported capabilities of each level format that was described in Section 3.1.

4.1.1 Level Format Capabilities

Every level format is expected to support a set of capabilities that can be invoked to access stored coordinates in a coordinate hierarchy level. Each capability is exposed as a set of *level functions* with a fixed interface, which a level format must implement in order to support the capability. Table 4.1 identifies the capabilities that are supported by each level format described in Section 3.1, and Table 4.2 shows how each level format implements the level functions that expose those capabilities.

Level format capabilities provide an abstraction for accessing sparse tensor storage in a format-agnostic manner. As an example, the column dimension of a CSR matrix is represented by a `compressed` level, which provides the coordinate position iteration capability. This capability is exposed as two level functions, `pos_bounds` and `pos_access`,

Table 4.2: Definitions of level functions that implement the supported capabilities of all the level formats described in Section 3.1.

Level Format	Level Function Definitions	

dense	coord_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {0, N_k}
		coord_access(..., p_{k-1}, ..., i_k):
		  return {p_{k-1} * N_k + i_k, true}

	locate(..., p_{k-1}, ..., i_k): return {p_{k-1} * N_k + i_k, true}	

masked	coord_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {0, N_k}
		coord_access(..., p_{k-1}, ..., i_k):
		  int p_k = p_{k-1} * N_k + i_k
		  return {p_k, mask[p_k]}

	locate(..., p_{k-1}, ..., i_k):
	  int p_k = p_{k-1} * N_k + i_k
	  return {p_k, mask[p_k]}	

sliced	coord_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {0, W_k}
		coord_access(..., p_{k-1}, ..., i_k):
		  return {p_{k-1} * W_k + i_k, true}

compressed	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {pos[p_{k-1}], pos[p_{k-1} + 1]}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {crd[p_k], true}

singleton	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {p_{k-1}, p_{k-1} + 1}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {crd[p_k], true}

squeezed	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {p_{k-1} * K_k, (p_{k-1} + 1) * K_k}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {perm[p_k - p_{k-1} * K_k], true}

range	coord_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {max(0, -i_{k-1}),
	  min(N_k, M_k - i_{k-1})}
		coord_access(..., p_{k-1}, ..., i_k):
		  return {p_{k-1} * N_k + i_k, true}

offset	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {p_{k-1}, p_{k-1} + 1}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {i_{k-1} + i_{k-2}, true}

block	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {p_{k-1}, p_{k-1} + 1}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {i_{k-2} + i_{k-4} * B_k, true}

ragged	coord_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {0, pos[p_{k-1} + 1] - pos[p_{k-1}]}
		coord_access(..., p_{k-1}, ..., i_k):
		  return {pos[p_{k-1}] + i_k, true}

	locate(..., p_{k-1}, ..., i_k): return {pos[p_{k-1}] + i_k, 0 <= i_k && i_k < (pos[p_{k-1} + 1] - pos[p_{k-1}])}	

hashed	pos_bounds(..., p_{k-1}, ..., i_{k-1}):
	  return {p_{k-1} * W_k, (p_{k-1} + 1) * W_k}
		pos_access(..., p_k, ..., i_{k-1}):
		  return {crd[p_k], crd[p_k] != -1}

	locate(..., p_{k-1}, ..., i_k):
	  int p_k = i_k % W_k + p_{k-1} * W_k
	  if (crd[p_k] != i_k && crd[p_k] != -1) {
	  int end = p_k
	  do {
	  p_k = (p_k + 1) % W_k + p_{k-1} * W_k
	  } while (crd[p_k] != i_k && crd[p_k] != -1 && p_k != end)
	  }
	  return {p_k, crd[p_k] == i_k}	

which `compressed` levels implement as shown in Table 4.2. To access the column coordinates that are highlighted in dark gray in Figure 3-2b, one can first determine the range of their positions by invoking `pos_bounds` with the position of row coordinate 3 as input. One then invoke `pos_access` for each position in this range to get the corresponding column coordinate. Under the hood, `pos_bounds` indexes the `pos` array to locate the `crd` array segment that stores the column coordinates, while `pos_access` retrieves each of the coordinates from `crd`. These level functions fully describe how to access CSR arrays efficiently while abstracting out the specifics from the caller. This, as we will see in Section 4.2, makes it possible for a compiler to reason about how to efficiently access and compute on tensors in different formats without having to hard-code for any format.

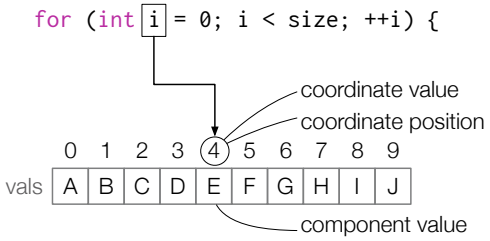
The abstract interface for array-based level formats exposes three distinct capabilities that capture how sparse tensor storage can be efficiently accessed: *coordinate value iteration*, *coordinate position iteration*, and *locate*. Every level format must support coordinate value iteration or coordinate position iteration, and a level format may optionally also support the locate capability. The rest of this subsection describes these capabilities in more detail.

Coordinate Value Iteration

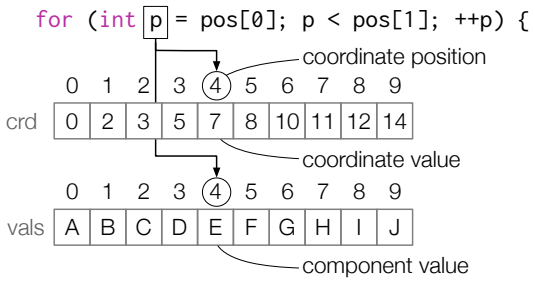
The coordinate value iteration capability directly iterates over coordinates in a coordinate hierarchy level. It generalizes the method in Figure 4-1a for iterating over a dense array. The capability is exposed as two level functions; the first effectively returns an iterator over coordinates that are stored in a coordinate hierarchy level (`coord_bounds`), while the second computes each coordinate's position in the level (`coord_access`):

```
coord_bounds(p_1, ..., p_{k-1}, i_1, ..., i_{k-1}) -> {ibegin_k, iend_k}
coord_access(p_1, ..., p_{k-1}, i_1, ..., i_k) -> {p_k, found}
```

More precisely, given ancestor coordinates i_1 to i_{k-1} encoded at positions p_1 (in the first level of the coordinate hierarchy) to p_{k-1} (in the $(i-1)$-th level), `coord_bounds` returns the bounds of an iterator over coordinates that may have those ancestors. For each coordinate i_k within those bounds, `coord_access` either returns the position of a child of i_{k-1} that encodes i_k and returns `found` as true, or alternatively it returns `found` as false if the coordinate does not actually exist. In the general case, these functions can be invoked



(a) Iterating over a dense array



(b) z Iterating over a sparse array

```
ibegin_k, iend_k = coord_bounds(...);
for (i_k = ibegin_k; i_k < iend_k; ++i_k) {
  p_k, found = coord_access(..., p_{k-1}, ..., i_k);
  if (found) {
    // coords and values dominated by i_k
    // at position p_k encode subtensor
    // B(i_1, ..., i_k, :, ..., :)
  }
}
```

(c) Coordinate value iteration

```
pbegin_k, pend_k = pos_bounds(...);
for (p_k = pbegin_k; p_k < pend_k; ++p_k) {
  i_k, found = pos_access(..., p_k, ..., i_{k-1});
  if (found) {
    // coords and values dominated by i_k
    // at position p_k encode subtensor
    // B(i_1, ..., i_k, :, ..., :)
  }
}
```

(d) Coordinate position iteration

Figure 4-1: To iterate over a dense vector, we loop over its *coordinates* and use them to index into the `vals` array. To iterate over a sparse vector, we loop over *coordinate positions* and use them to access the `crd` and `vals` arrays. Coordinate value iteration and coordinate position iteration generalize these patterns to other array-based data structures.

to iterate over coordinates of tensor elements as demonstrated in Figure 4-1c. In practice though, the code in Figure 4-1c can be optimized by omitting the conditional if we statically analyze the definition of `coord_access` and determine that it always returns `found` as true.

Coordinate Position Iteration

The coordinate position iteration capability, on the other hand, iterates over positions of coordinates in a coordinate hierarchy level. It generalizes the method in Figure 4-1b for iterating over a sparse array. Again, the capability is exposed as two level functions; the first effectively returns an iterator over positions in a coordinate hierarchy level (`pos_bounds`) and the second accesses the coordinate encoded at each position (`pos_access`):

```
pos_bounds(p_1, ..., p_{k-1}, i_1, ..., i_{k-1}) -> {pbegin_k, pend_k}
pos_access(p_1, ..., p_k, i_1, ..., i_{k-1}) -> {i_k, found}
```

More precisely, given ancestor coordinates i_1 to i_{k-1} encoded at positions p_1 (in the first level of the coordinate hierarchy) to p_{k-1} (in the $(i-1)$-th level), `pos_bounds` returns the

bounds of an iterator over positions that may have a coordinate with parent at position p_{k-1}. For each position p_k within those bounds, `pos_access` either returns the coordinate encoded at that position and returns `found` as true, or alternatively it returns `found` as false if there is no valid coordinate encoded at p_k or if the coordinate at p_k is not actually a child of the coordinate at p_{k-1}. These functions can be invoked to iterate over coordinates of tensor elements as shown in Figure 4-1d; this code shares a similar structure as code for coordinate value iteration, though with the roles of i_k and p_k reversed.

Locate

The locate capability provides constant-time random access into a coordinate hierarchy level through a function that computes the position of a coordinate:

```
locate(p_1, ..., p_{k-1}, i_1, ..., i_k) -> {p_k, found}
```

The `locate` function has similar semantics as `coord_access`. In particular, given a coordinate i_{k-1} encoded at position p_{k-1}, `locate` attempts to find among its children the coordinate i_k. If `locate` finds i_k, then it returns the coordinate's position p_k and returns `found` as true; otherwise, it returns `found` as false. Traversing a path in a coordinate hierarchy in order to access a single tensor element can be done by successively invoking `locate` at every level. As we will see in Section 4.2, having operands with efficient implementations of the locate capability leads to code that avoids having to iterate over every stored element in those operands, which can improve compute performance.

4.1.2 Level Format Properties

Coordinate hierarchy levels may also possess up to six distinct *properties*: *full*, *ordered*, *unique*, *zeroless*, *branchless*, and *compact*. These properties describe characteristics of coordinates that are encoded in a level, such as whether or not the coordinates are arranged in order. The column dimension of a sorted CSR matrix, for instance, is both ordered and unique (Figure 3-3), which means that it stores each column coordinate in a row at most once and that it stores those coordinates in increasing order. As we will see in

Section 4.2, our technique can generate code that exploit such level properties of sparse tensor operands in order to improve compute performance.

Properties of coordinate hierarchy levels are invariants that are either explicitly enforced or implicitly assumed by the data structures storing the levels. Table 4.1 identifies the properties that are enforced or assumed by each level format described in Section 3.1. Some level formats may optionally declare certain properties depending on the application; such optional properties reflect invariants that are not tied to how the underlying data structures store coordinates. For example, the `crd` array in `compressed` levels typically store coordinates in order when used in the CSR format, but the same data structure can also store coordinates out of order. Figure 3-3 shows several examples of how level formats with optional properties can be configured to represent distinct tensor formats.

The rest of this subsection describes the six level properties in more detail.

Full

A coordinate hierarchy level is full if every collection of coordinates that share the same ancestors encompasses all valid coordinates along the corresponding tensor dimension. For example, a level that represents a CSR matrix's row dimension (Figure 3-2b) encodes every row coordinate and is thus full. By contrast, a level that represents the same CSR matrix's column dimension is not full, since it only stores the coordinates of nonzeros.

Unique

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

Ordered

A coordinate hierarchy level is ordered if coordinates that share the same ancestors are ordered in increasing value, coordinates with different ancestors are ordered lexicographi-

cally by their ancestors, and duplicates are ordered by their parents' positions. For example, a level that represents a sorted CSR matrix's column dimension is ordered, since it stores column coordinates in the same row in increasing order. By contrast, a level that represents the column dimension of a hash map row vector (Figure 3-1c) is not ordered, since coordinates are stored in hash order instead.

Zeroless

A coordinate hierarchy level is zeroless if every coordinate corresponds to a nonzero or a non-empty slice of the tensor (i.e., if the coordinate is the ancestor of at least one leaf coordinate that corresponds to a nonzero). For example, a level that represents a fully-compressed CSR matrix's column dimension is zeroless, since it only stores column coordinates that correspond to nonzeros in the matrix. By contrast, a level that represents a DIA matrix's column dimension (Figure 3-2e) would not be zeroless if some of the column coordinates correspond to zero padding.

Branchless

A coordinate hierarchy level is branchless if every coordinate in the parent level has exactly one child node and each such child node encodes a coordinate. For example, the coordinate hierarchy representation of a COO matrix consists strictly of chains of coordinates, making the lower level branchless. By contrast, a level that represents a CSR matrix's column dimension can have multiple coordinates with the same parent and is thus not branchless.

Compact

A coordinate hierarchy level is compact if no two coordinates are separated by an unlabeled node that does not encode a coordinate. For instance, a level that represents a CSR matrix's column dimension encodes coordinates in one contiguous range of positions and is thus compact. By contrast, a level that represents the column dimension of a hash map row vector is not compact, since it can have unlabeled positions that reflect empty buckets.

4.2 Code Generation

I now describe how, guided only by the abstract interface described in the previous section, a compiler can generate efficient sparse tensor algebra code to compute on operands that may be stored in arbitrary combinations of array-based sparse tensor formats. Our technique generalizes that of Kjolstad et al. [54, 53], which only supported sparse tensor formats that store nonzeros using the same arrays as CSR, to handle many more disparate formats by only reasoning about their properties and supported capabilities. This approach decouples a compiler's complexity from the number of supported formats, which makes it possible for the compiler to support many disparate formats without sacrificing maintainability or the performance of generated code.

4.2.1 Background

The code generation algorithm of Kjolstad et al. [54, 53] takes as input a tensor algebra computation that is expressed in *concrete index notation*, which specifies how each element in the output tensor should be computed in terms of elements in the input tensors. Matrix addition, for example, can be expressed in concrete index notation as $\forall_i \forall_j \; A_{ij} = B_{ij} + C_{ij}$, which specifies that each element in the result is the sum of the corresponding elements in the input tensors. Similarly, matrix multiplication can be expressed as $\forall_i \forall_j \forall_k \; A_{ij} \mathrel{+}= B_{ik} C_{kj}$, which makes explicit that each element A_{ij} in the result is the inner product of the i-th row of B and the j-th column of C.

Computing an expression in concrete index notation requires merging its operands—that is to say, iterating over the joint iteration space of the operands—dimension by dimension. For instance, code to add sparse matrices must iterate over rows that have nonzeros in either matrix and, for each such row, iterate over elements that are non-zero in either matrix. Additive computations must iterate over the union of the operand nonzeros (i.e., a union merge), while multiplicative computations must iterate over the intersection of the operand nonzeros (i.e., an intersection merge). The code generation algorithm of Kjolstad el al. generates code to merge operands by recursively emitting one or more loops to iterate over each dimension in the joint iteration space. So to generate code that

(a) Merge lattice for i dimension

(b) Merge lattice for j dimension

Figure 4-2: Optimized merge lattices for sparse matrix addition ($\forall_i \forall_j\ A_{ij} = B_{ij} + C_{ij}$), where B is a CSR matrix and C can either be a CSR matrix or a COO matrix that is guaranteed to contain no empty row.

computes matrix addition, for instance, the code generator first emits one or more loops to iterate over the union of non-empty rows in B and C (i.e., dimension i). Then, within each emitted loop over the non-empty rows in B and C, the code generator further emits one or more loops to iterate over the union of nonzeros within corresponding rows in B and C (i.e., dimension j) in order to compute the desired summation.

For each dimension—indexed by some index variable v—in the joint iteration space, its corresponding *merge lattice* describes what loops are actually needed in order to merge all input tensor dimensions that are indexed by v. Each point in the merge lattice encodes a set of input tensor dimensions indexed by v that may contain nonzeros and that need to be simultaneously merged in one loop. Each lattice point also encodes a sub-expression that needs to be computed in the corresponding loop. 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 tensors. Figures 4-2a and 4-2b, for instance, show merge lattices that describe the addition of two CSR matrices. To fully merge B and C's column dimensions, we would start by running the loop that corresponds to the top lattice point in Figure 4-2b, computing $A_{ij} = B_{ij} + C_{ij}$ in each iteration. This incrementally merges the two operands until one (e.g., B) has been fully merged into the output. Then, to also fully merge the other operand (i.e., C) into the output, we would only have to run the loop that corresponds to the middle-left lattice point in Figure 4-2b, which computes $A_{ij} = C_{ij}$ in each iteration in order to copy the remaining nonzeros from C to the output.

4.2.2 Property-Based Merge Lattice Optimizations

Kjolstad et al. [54] describe how merge lattices can be simplified in order to yield optimized code. However, these optimizations were all hard-coded to `dense` and `compressed` level formats. To generalize the optimizations so that they can be applied to any level format, we reformulate the optimizations with respect to properties and supported capabilities of coordinate hierarchy levels. In particular, given a merge lattice for index variable v, our algorithm removes any lattice point that does not merge every full tensor dimension (i.e., a dimension represented by a full coordinate hierarchy level) that v indexes into. This optimization is valid since full tensor dimensions are supersets of any sparse dimension, so once we finish iterating over and merging a full dimension we must have also visited every coordinate in the joint iteration space. Applying this optimization gives us the optimized merge lattice shown in Figure 4-2a, which contains just a single lattice point even though the operation being computed requires a union merge.

Our technique also optimizes merging of any number of full tensor dimensions by emitting code to simultaneously iterate (i.e., co-iterate) over only those that do not support the locate capability, with the rest accessed via calls to `locate`. Depending on whether the operands are unordered, this can reduce the complexity of the co-iteration and thus the merge, assuming `locate` runs in constant time.

4.2.3 Merging Coordinate Hierarchy Levels

Merging dimensions of tensor operands can be equivalently viewed as merging the coordinate hierarchy levels that represent those dimensions. The most efficient method for merging levels depends on the properties and supported capabilities of the merged levels. Consider, for instance, the element-wise multiplication of two vectors x and y, which requires iterating over the intersection of coordinate hierarchy levels that encode their nonzeros. Figure 4-3 shows the asymptotically most efficient strategies for computing the intersection merge depending on whether the input tensors are ordered or unique and whether they support the locate capability. These are the same strategies that our code generation algorithm, which I describe in depth in Section 4.2.5, selects.

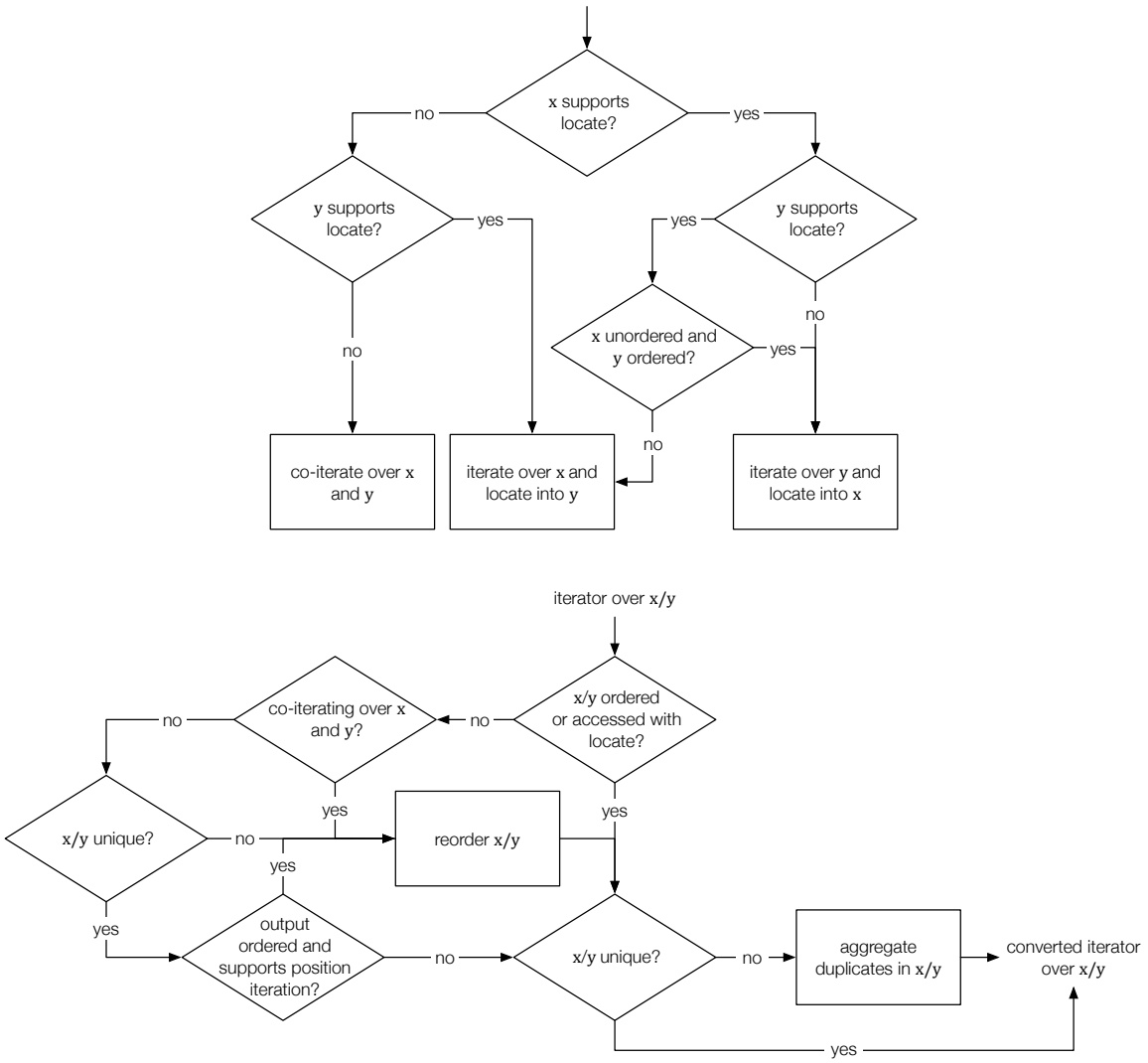


Figure 4-3: The most efficient strategies for computing the intersection merge of two vectors x and y, depending on whether they support the locate capability and whether they are ordered and unique. The sparsity structure of y is assumed to not be a strict subset of the sparsity structure of x. The flowchart on the right describes, for each operand, what iterator conversions are needed at runtime to compute the merge.

If neither input vector supports the locate capability, we can simultaneously iterate (i.e., co-iterate) over the coordinate hierarchy levels that represent those vectors and compute a new output element whenever we encounter nonzeros in both inputs that share the same coordinate. Lines 8–20 in Figure 1-4c shows another example of this method applied to merge the column dimensions of a CSR matrix and a COO matrix. Co-iterating over multiple coordinate hierarchy levels does require the ability to enumerate stored coordinates uniquely and in order. Even so, this method can be used to merge

coordinate hierarchy levels that are non-unique or unordered as well by applying run-time transformations (as specified in Figure 4-3) to obtain iterators that nevertheless enumerate stored coordinates uniquely and in order. Section 4.2.4 describes these run-time transformations (i.e., iterator conversions) in depth.

If one of the input vectors, say y, supports the locate capability (e.g., it is stored in a dense array) however, we can instead just iterate over the non-zero elements of x and, for each nonzero, locate the corresponding element with the same coordinate in y. Lines 2–9 in Figure 1-4b shows another example of this method applied to merge the column dimensions of a CSR matrix and a dense matrix. This alternative method reduces the asymptotic complexity of the merge from $O(\text{nnz}(x) + \text{nnz}(y))$ to $O(\text{nnz}(x))$ assuming `locate` runs in constant time. In addition, this method does not require enumerating the coordinates of y in order. Moreover, we do not even need to enumerate the coordinates of x in order if there are no duplicates and we do not need to compute elements of the result in order (i.e., if elements of the output tensor are not stored in order, or if the output tensor supports coordinate value iteration). This method is thus ideal for computing intersection merges of unordered coordinate hierarchy levels.

We can generalize and combine the two methods described above to compute arbitrarily complex merges involving unions and intersections of any number of tensor operands. At a high level, any merge can be computed by co-iterating over some subset of its operands and, for every enumerated coordinate, locating that same coordinate in all the remaining operands with calls to `locate`. Which operands need to be co-iterated can be identified recursively from the expression *expr* that we want to compute. In particular, for each subexpression $e = e_1 \ op \ e_2$ in *expr*, let $Coiter(e)$ denote the set of operands that need to be co-iterated in order to compute e. If *op* is an operation that requires a union merge (e.g., addition), then computing e requires co-iterating over all the operands that would have to be co-iterated in order to separately compute e_1 and e_2; in other words, $Coiter(e) = Coiter(e_1) \cup Coiter(e_2)$. On the other hand, if *op* is an operation that requires an intersection merge (e.g., multiplication), then the set of non-zero elements in the result e must be a subset of non-zero elements in either e_1 or e_2. Thus, in order to enumerate the coordinates of all nonzeros in the result, it suffices to just co-iterate over the operands that

must be merged by e_1 or e_2. Without loss of generality, this lets us compute e without having to co-iterate over operands merged by e_2 that can instead be accessed with `locate`; in other words, $Coiter(e) = Coiter(e_1) \cup (Coiter(e_2) \setminus LocateCapable(e_2))$, where $LocateCapable(e_2)$ denotes the set of operands merged by e_2 that support the locate capability.

4.2.4 Iterator Conversion

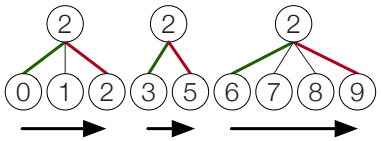
As we saw in the previous subsection, efficient algorithms exist for merging ordered and unique coordinate hierarchy levels as well as for computing intersection merges involving unordered levels that provide the locate capability. To also support computing arbitrary merges of unordered or non-unique coordinate hierarchy levels though, we need to be able to obtain iterators that enumerate stored coordinates in such levels uniquely and in order.

Iterator conversion refers to a set of run-time transformations that on-the-fly convert iterators over unordered or non-unique coordinate hierarchy levels to new iterators with the desired properties. The rest of this subsection describes two types of iterator conversion—*deduplication* and *reordering*—that can be composed as needed to support arbitrary merges of any coordinate hierarchy levels. In particular, the flowchart on the right in Figure 4-3 identifies, for each operand in an intersection merge, the specific iterator conversions that are needed to merge the operand. In the next subsection, I show how our code generation algorithm can emit code to perform these necessary iterator conversions at run time.

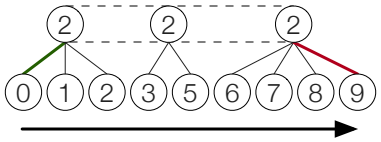
Deduplication

Duplicate coordinates complicate merging because it can result in the same points in the iteration space being repeatedly visited. Iterator deduplication removes duplicate coordinates from iterators over ordered and non-unique coordinate hierarchy levels using a deduplication loop. Lines 5–7 in Figure 1-4c shows an example of a deduplication loop that scans ahead and aggregates nonzeros with duplicate coordinates, resulting in an iterator that enumerates coordinates uniquely.

When non-unique levels are at the bottom of coordinate hierarchies, our technique emits deduplication loops that sum the values corresponding to the duplicate coordinates.



(a) Separate iterators over the children of duplicate coordinates



(b) Chained iterator over the children of duplicate coordinates

Figure 4-4: Iterator chaining chains the iterators over the children of duplicate coordinates (a) into a single iterator over all the children (b). The arrows represent iterators with start and end bounds as green and red edges.

Otherwise, the emitted deduplication loop combines iterators over the duplicate coordinates' children into a single iterator. In general, this requires assembling a scratch array to store the child coordinates in order. Figure 4-4 shows, though, how one can avoid the need for a scratch array by logically chaining together iterators over the children. With iterator chaining, the starting bound of the first set of children and the ending bound of the last set of children become the bounds of the chained iterator. The resulting iterator provides the same interface as a regular coordinate position iterator and can thus participate in merging without a scratch array. Figure 1-4c shows how iterator chaining can be used to iterate over columns of a COO matrix. This optimization, however, requires the child and parent coordinate hierarchy levels to both be ordered and compact. Additionally, the optimization requires the child level to be stored in a format that supports coordinate position iteration, and the level format's implementation of `pos_access` cannot use the position of the parent coordinate (i.e., p_{k-1}) to access child coordinates.

Reordering

A necessary precondition for co-iterating over coordinate hierarchy levels is that it must be possible to enumerate the stored coordinates in order. However, this might not be possible if a coordinate hierarchy level is unordered and only supports coordinate position iteration. Iterator reordering addresses this by assembling scratch arrays that store an ordered copy of each unordered coordinate hierarchy level and by replacing iterators over the unordered levels with iterators over their ordered copies. Unordered coordinate hierarchy levels can then be merged by co-iterating over their ordered copies instead.

4.2.5 Code Generation Algorithm

Figure 4-5 shows our code generation algorithm, which utilizes all of the ideas described in the previous subsections to generate efficient sparse tensor algebra code. Each part of the algorithm is labeled from 1 to 11; throughout the discussion of the algorithm in the rest of this subsection, I identify relevant parts using these labels.

The algorithm shown in Figure 4-5 emits code that iterates over the proper intersections and unions of the input tensors by invoking relevant level functions. Our technique then specializes the emitted code to compute with tensors that are stored in specific formats by mechanically inlining all level function calls. This approach bounds the complexity of the code generation mechanism, since it only needs to account for a finite and fixed set of level format capabilities and properties. The result is an algorithm that naturally supports many disparate sparse tensor formats and that can be extended to support even more formats without requiring users to modify the compiler itself. Figure 4-6 shows an example of code that our algorithm generates, with level function calls inlined.

Our algorithm takes as input a concrete index notation statement and, in the order specified by the ∀s, recursively invokes itself on index variables in the statement. At each level of recursion, the algorithm generates code to iterate over the dimension that is indexed by `index-var` in the joint iteration space. The algorithm begins by emitting code that initializes iterators over the coordinate hierarchy levels representing input tensors, which entails calling their appropriate coordinate value or position iteration level functions (1). The algorithm also emits code to perform any necessary iterator conversion as described in Section 4.2.4 (1, 2).

The algorithm additionally constructs a merge lattice at every level of recursion for the dimension indexed by `index-var`. This is done by applying the merge lattice construction algorithm proposed by Kjolstad et al. [54, Section 5.1] and simplifying the resulting lattice with the first optimization described in Section 4.2.2. For every point L_p in the simplified merge lattice, the algorithm then emits a loop to merge the coordinate hierarchy levels representing input tensor dimensions that need to be merged by L_p (4). The subset of merged levels that must be co-iterated by each loop (i.e., `coiter-dims`(L_p)) is determined

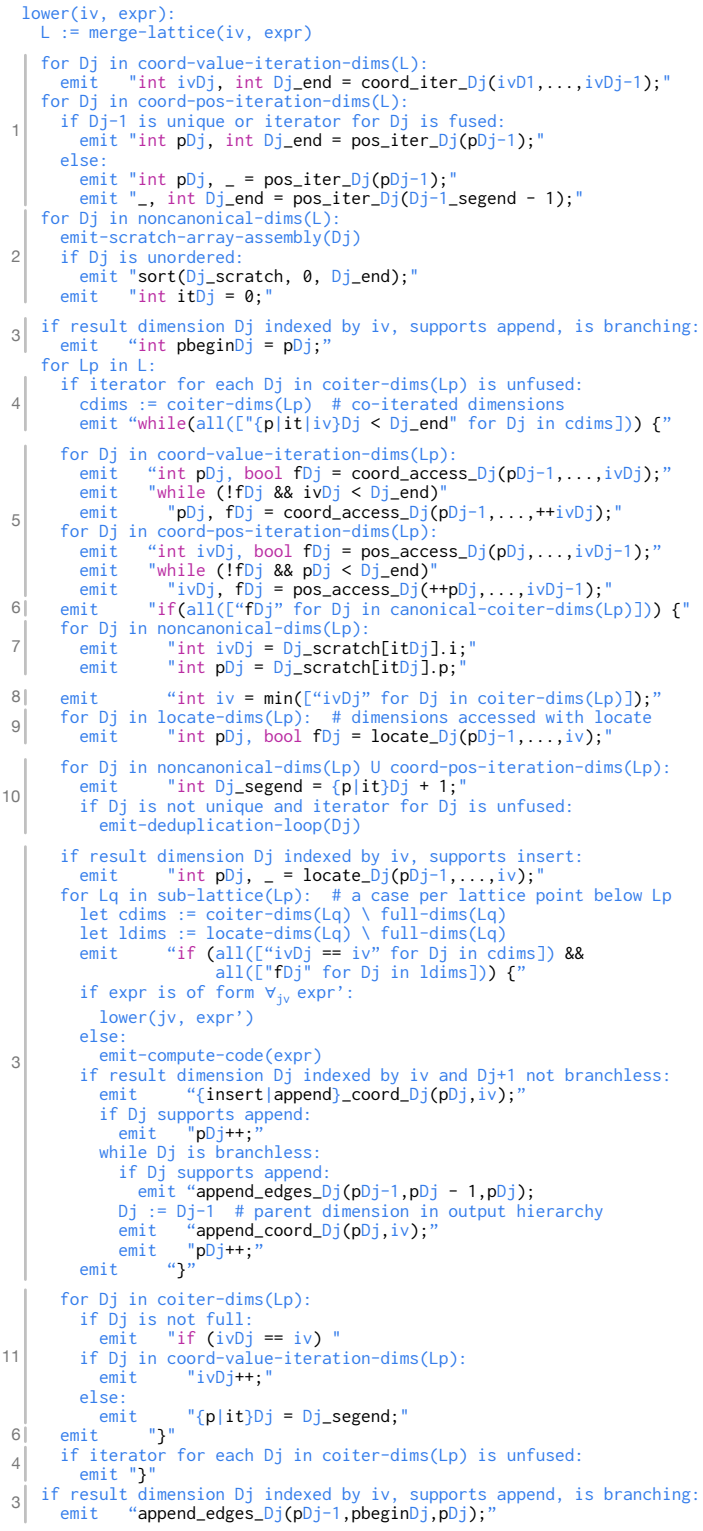


Figure 4-5: Algorithm for generating code that computes the concrete index notation statement $\forall_{iv}$ *expr* on operands stored in array-based formats by invoking their level function implementations. The sets `coord-value-iteration-dims` and `coord-pos-iteration-dims` exclude dimensions in `noncanonical-dims`, which are those that require a scratch array (as described in Section 4.2.4) in order to be co-iterated.

```
1  int pC1 = C1_pos[0];
   int C1_end = C1_pos[1];
4  while (pC1 < C1_end) {
5    int iC1 = C1_crd[pC1];
8    int i = iC1;
9    int pB1 = (0 * B1_N) + i;
     int C1_segend = pC1 + 1;
10   while (C1_segend < C1_end &&
            C1_crd[C1_segend] == i)
       C1_segend++;
3    int pA1 = (0 * A1_N) + i;
     int pB2 = B2_pos[pB1];
1    int B2_end = B2_pos[pB1 + 1];
     int pC2 = pC1;
     int C2_end = C1_segend;
4    while (pB2 < B2_end &&
            pC2 < C2_end) {
5      int jB2 = B2_crd[pB2];
       int jC2 = C2_crd[pC2];
8      int j = min(jB2, jC2);
10     int B2_segend = pB2 + 1;
       int C2_segend = pC2 + 1;
       int pA2 = (pA1 * A2_N) + j;
       if (jB2 == j && jC2 == j) {
         A[pA2] = B[pB2] + C[pC2];
3      } else if (jB == j) {
         A[pA2] = B[pB2];
       } else if (jC == j) {
         A[pA2] = C[pC2];
       }
11     if (jB2 == j) pB2 = B2_segend;
       if (jC2 == j) pC2 = C2_segend;
4    }
5    while (pB2 < B2_end) {
8      int jB2 = B2_crd[pB2];
       int j = jB2;
10     int B2_segend = pB2 + 1;
3      int pA2 = (pA1 * A2_N) + j;
       A[pA2] = B[pB2];
11     pB2 = B2_segend;
4    }
5    while (pC2 < C2_end) {
8      int jC2 = C2_crd[pC2];
       int j = jC2;
10     int C2_segend = pC2 + 1;
3      int pA2 = (pA1 * A2_N) + j;
       A[pA2] = C[pC2];
11     pC2 = C2_segend;
4    }
11   pC1 = C1_segend;
4  }
```

Figure 4-6: Code that our technique generates—with level function calls inlined—for adding a CSR matrix and a COO matrix with no empty row, with the result stored in a dense array.

by applying the recursive algorithm described in Section 4.2.3 (with the subexpression to be computed by L_p as input) and applying the second optimization described in Section 4.2.2. Within each loop, the generated code dereferences (potentially converted) iterators over the levels that must be co-iterated (5, 7, 10), making sure to not inadvertently dereference any iterator that has exceeded its ending bound (6). The next coordinate to be visited in

the joint iteration space, iv, is then computed (8) and used to index into the levels that can instead be accessed with the locate capability (9). At the end of each loop iteration, the generated code advances every iterator that referenced the merged coordinate iv (11), so that subsequent iterations of the loop will not visit the same coordinate again.

Within each loop, the generated code must also actually compute the value of the result tensor at each coordinate as well as assemble the output data structures for storing those values (3). The algorithm emits specialized compute and assembly code for each merge lattice point that is dominated by L_p, which handles the case where the corresponding subset of inputs contain nonzeros at the same coordinate. Chapter 6 describes how our technique emits assembly code in more depth.

Fusing Iterators By default, at every level of recursion, our 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 like COO entails emitting code that simultaneously iterates over multiple coordinate hierarchy levels of one tensor. Our algorithm implements this optimization by fusing iterators over branchless levels with iterators over their preceding levels. This is legal as long as the fused iterators do not need to participate in co-iteration (i.e., if the other levels to be merged can be accessed with the locate capability). The algorithm then avoids emitting loops for levels accessed by fused iterators (4), which eliminates unnecessary branching overhead. For some computations, however, this optimization transforms the emitted kernel from gather code that enumerates each result nonzero once to scatter code that accumulates into the output. In such cases, our algorithm ensures that the output also supports the locate capability. Figure 1-4a gives an example of code that our technique generates with this optimization, which iterates over two tensor dimensions with a single loop.

4.3 Chapter Summary

In this chapter, we saw how array-based level formats can be precisely defined by implementing an abstract interface that exposes properties of the stored data and that captures

how the stored data can be efficiently accessed. We further saw how a compiler can use implementations of this abstract interface to generate efficient code for computing on sparse tensors that are stored in array-based level formats. As we will see, however, such an approach does not readily generalize to pointer-based level formats. In the next chapter, we will see how pointer-based level formats can be precisely defined by instead specifying how the underlying pointer-based data structures organize stored elements in memory. Meanwhile, in Chapter 6, we will see how our technique can also generate efficient code for storing the results of sparse tensor operations in array-based level formats.

Chapter 5

Supporting Pointer-Based Formats

In the previous chapter, we saw how, by utilizing implementations of level functions that define how tensors stored in different array-based formats can be efficiently accessed, a compiler can generate efficient code to compute on tensors stored in those formats. Unfortunately, while such an approach suffices for supporting array-based tensor formats, the same approach falls short in supporting pointer-based tensor formats for several reasons. For one thing, as the example in Figure 1-5c demonstrates, iterators that efficiently enumerate the stored elements of pointer-based data structures like BSTs are often much more sophisticated than equivalent iterators for array-based data structures. Thus, requiring users to correctly implement level functions that describe how to efficiently iterate over pointer-based formats would make the compiler much more difficult for non-expert users to use. For another, even if users can manually implement efficient iterators for pointer-based formats, as the example in Figure 1-5b illustrates, such iterators alone would not be enough to support efficiently computing all possible sparse tensor computations.

In this chapter, I describe how we generalize our technique to support pointer-based level formats, which store coordinates of nonzeros using pointer-based data structures such as linked lists and BSTs. In particular, I first show how pointer-based level formats can be precisely defined using a language we developed that we call the *node schema language*, which lets users specify exactly how the underlying data structures organize stored elements in memory (Section 5.1). I then describe how, guided by such specifications of pointer-based level formats, a compiler can generate efficient code to compute tensor

algebra operations on operands that may be stored in arbitrary combinations of pointer-based sparse tensor formats (Section 5.2). Again, each pointer-based level format can be defined completely independently of other formats, which makes it practical to add support for a large number of formats. And by again decoupling specifications of formats from the code generation mechanism, our approach makes it possible to add support for new pointer-based formats without modifying the compiler itself. Furthermore, by relying on specifications that do not implement specific algorithms for accessing pointer-based data structures, a compiler can use the same specifications to generate code that access the same sparse tensors in very different ways depending on what computation needs to be performed. The result is a compiler that, without sacrificing ease of use, can achieve good performance across a wide range of computations on sparse tensors that are stored in a wide range of formats.

5.1 Node Schema Language

A wide range of pointer-based data structures, including all those described in Section 2.2, can be modeled as collections of nodes that are stored non-contiguously in memory, with each node storing a subset of elements. To precisely define a pointer-based level format that uses any one of the aforementioned data structures to store coordinates of nonzeros, our technique requires a user to provide *schemas* of the data structure's nodes, which specify how stored coordinates are distributed amongst nodes and how nodes are linked together. These schemas can be expressed using the node schema language, the syntax for which is provided in Figure 5-1.

The node schema language allows users to define nodes that can contain an arbitrary number of fields, each of which may store coordinates of nonzeros or store references to other nodes. As an example, Figure 5-2 shows how binary search trees can be precisely defined using the node schema language. In particular, a binary search tree consists of two types of nodes: a `bst_root` node, which simply stores a reference to the root of the tree, and `bst` nodes, which actually contain the nonzeros. The schema for `bst` nodes specifies that each node stores a coordinate `e` (and its corresponding nonzero or non-empty subtensor)

⟨*node_schema*⟩ ::= ⟨*supertype_def*⟩* ⟨*node_def*⟩+

⟨*supertype_def*⟩ ::= 'def' 'supertype' ⟨*name*⟩

⟨*node_def*⟩ ::= 'def' ⟨*name*⟩ [':' ⟨*name*⟩] '{' ⟨*field_def*⟩+ [⟨*sequence_def*⟩] '}'

⟨*field_def*⟩ ::= ⟨*name*⟩ ':' ⟨*type*⟩

⟨*type*⟩ ::= ⟨*elem_type*⟩ | ⟨*child_type*⟩ | ⟨*size_type*⟩ | ⟨*metadata_type*⟩ | 'parent'

⟨*elem_type*⟩ ::= 'elem' [⟨*array_type*⟩] ['nonempty']

⟨*child_type*⟩ ::= ⟨*name*⟩ [⟨*array_type*⟩] ['nonempty']

⟨*array_type*⟩ ::= '[' (⟨*name*⟩ | ⟨*const*⟩) ']'

⟨*size_type*⟩ ::= 'size' ['in' ⟨*array_size*⟩]

⟨*array_size*⟩ ::= '[' ⟨*const*⟩ ',' (⟨*const*⟩ | '*') ']'

⟨*metadata_type*⟩ ::= 'bool' | 'int8' | 'uint8' | 'int16' | 'uint16' ...

⟨*sequence_def*⟩ ::= 'seq' '=' ⟨*seq_entry*⟩ (',' ⟨*seq_entry*⟩)*

⟨*seq_entry*⟩ ::= ⟨*name*⟩ | '{' ⟨*name*⟩ (',' ⟨*name*⟩)* '}'

Figure 5-1: Syntax of the node schema language.

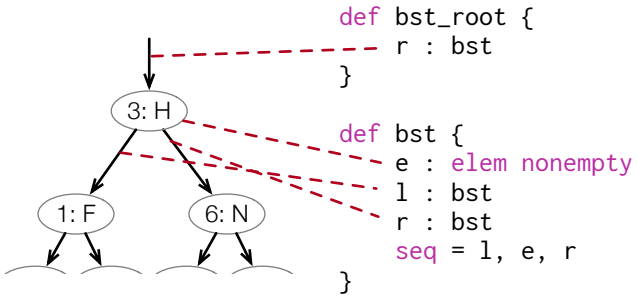


Figure 5-2: The node schemas for a BST precisely specifies how the coordinates (and values) of nonzeros are stored in nodes of a BST and how these nodes are linked together.

as well as stores references to up to two child nodes `l` and `r`, both of which are of the same type. (The `nonempty` annotation specifies that each node must store exactly one coordinate and cannot be empty.) Furthermore, the schema contains a *sequence attribute* (`seq`) that specifies the ordering of coordinates stored by all reachable nodes; in particular, all coordinates reachable from `l` must be smaller than `e`, which in turns must be smaller than all coordinates reachable from `r`. Meanwhile, the schema for the `bst_root` node specifies that it stores a reference to the root node `r`, which may be null if the tree is empty.

The node schema language assumes, by default, that any data structure being defined is acyclic. This means that, for instance, while a `bst` node may store references to child nodes

of the same type, it cannot store a reference to an ancestor node as its child. (That said, as Figure 5-3e demonstrates, one can define a variant of BSTs that stores a reference to each node's parent in a `parent` field. As the example also shows though, a node's sequence attribute cannot constrain the ordering of coordinates that are stored in a parent node.)

Nodes in a pointer-based data structure may be defined to store more than one nonzero. Figure 5-3d, for instance, shows how T-trees [63], which generalize BSTs by having each node store a bounded-size block of elements, can be precisely defined. In particular, the schema for `ttree` nodes specifies that each node can store multiple coordinates (and their corresponding nonzeros or non-empty subtensors) contiguously in an array `e`, with the exact number of nonzeros that `e` contains being stored in a separate field `B`. Different nodes may store different numbers of nonzeros, but the `in` clause (in the declaration of the `B` field) constrains each node to contain at least one and at most four nonzeros. Like with BSTs, the sequence attribute specifies that all coordinates stored in a node (in array `e`) are larger than all coordinates that are stored in the left subtree `l` but smaller than all coordinates that are stored in the right subtree `r`. Additionally though, the `{e}` term in the sequence attribute indicates that coordinates are stored within `e` in increasing order. This means that `e[0]` stores the smallest coordinate, `e[1]` stores the second-smallest coordinate, and so on. More generally, a sequence attribute term that is enclosed within braces may reference multiple arrays, which specifies that the array elements are ordered in interleaving order. So, for instance, the term `{e, cr}` in the sequence attribute for internal nodes of B-trees (Figure 5-3g) denotes that `e[0]` is smaller than all coordinates that are stored in the subtree `cr[0]`, which in turn are all smaller than `e[1]`, and so on.

Annotations to node schemas and their fields, including `nonempty` annotations as well as sequence attributes, are strictly optional, making it possible to define many practical variants of a pointer-based data structure. Figures 5-3b and 5-4, for instance, show how the node schema language can be used to define four variants of block linked lists, each of which pads blocks and orders stored coordinates in a different way. Similarly, Figure 5-3f shows how a declaration of a `size` field can omit the `in` clause, indicating that the size of an array field is unconstrained. This makes it possible to precisely define a C-tree, which, unlike T-trees, does not strictly limit the number of coordinates stored in each node.

```
def list {
  e : elem nonempty
  n : list
  seq = {e}, n
}

def list_head {
  h : list
}
```

(a) Linked list

```
def blist {
  e : elem[B] nonempty
  n : blist
  B : size in [0, 3]
  seq = {e}, n
}

def blist_head {
  h : blist
}
```

(b) Block linked list

```
def vblist {
  e : elem[B] nonempty
  n : vblist
  B : size
  seq = {e}, n
}

def vblist_head {
  h : vblist
}
```

(c) Variable block linked list

```
def ttree {
  e : elem[B] nonempty
  l : ttree
  r : ttree
  B : size in [1, 5]
  seq = l, {e}, r
}

def ttree_root {
  r : ttree
}
```

(d) T-tree

```
def rbtree {
  e : elem nonempty
  l : rbtree
  r : rbtree
  p : parent
  c : bool
  seq = l, e, r
}

def rbtree_root {
  r : rbtree
}
```

(e) Red-black tree

```
def tree {
  h : elem nonempty
  t : chunk
  l : tree
  r : tree
  seq = l, h, t, r
}

def chunk {
  e : elem[N] nonempty
  N : size
  seq = {e}
}

def ctree {
  p : chunk
  r : tree
  seq = p, r
}
```

(f) C-tree

```
def supertype btree

def btree_internal : btree {
  e  : elem[B] nonempty
  cf : btree nonempty
  cr : btree[B] nonempty
  B  : size in [1, 3]
  seq = cf, {e, cr}
}

def btree_leaf : btree {
  e : elem[B] nonempty
  B : size in [1, 3]
  seq = {e}
}

def btree_root {
  r : btree
}
```

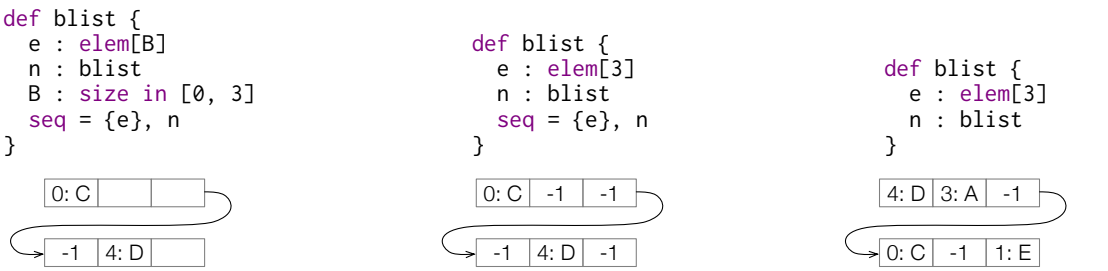
(g) B-tree

```
def hybrid {
  e : elem[B] nonempty
  r : btree
  B : size in [0, 5]
  seq = {e}, r
}
```

(h) Fixed-size array/B-tree hybrid

Figure 5-3: Node schemas for a wide range of pointer-based data structures, including all those shown in Figure 2-4.

A pointer-based data structure may further be defined to consist of multiple types of nodes that store coordinates of nonzeros in different ways. Figure 5-3f demonstrates, for instance, how C-trees can be expressed in the node schema language by defining two types of nodes—`tree` and `chunk`—for storing coordinates. In particular, `tree` nodes organize all of the head elements into a BST, with each node storing a single head element in its `h` field. Meanwhile, each `chunk` node uses a single array `e` to store either the C-tree's prefix or all tail elements that correspond to a particular head element.



(a) With possibly empty slots up to position `B` in each block

(b) With possibly empty slots anywhere in each block

(c) With stored coordinates unsorted

Figure 5-4: The node schema language can describe many variants of block linked lists. In the subfigures above, unlabeled slots are those that, at run time, can automatically be assumed to not store any nonzero (i.e., must be empty) and can therefore be skipped when iterated over. Slots labeled "-1", on the other hand, might store nonzeros (i.e., are only possibly empty) and must therefore be explicitly checked when iterated over.

While different types of nodes may possess different sets of fields, they can nevertheless share a common supertype, which allows a single reference to point to a node that is of one of several different types. For instance, B-trees consist of two types of nodes: internal nodes, which need to store references to child nodes, and child nodes, which can omit those references to reduce space usage. As Figure 5-3g shows, by defining internal nodes (`btree_internal`) and leaf nodes (`btree_leaf`) to be of the same supertype `btree`, a user can specify that each child of an internal node may itself be another internal node or, alternatively, be a leaf node.

Finally, the node schema language allows users to specify that nodes store additional metadata, which may not be strictly needed to store coordinates of nonzeros but are useful for other purposes. For instance, Figure 5-3e shows how a node in a red-black tree can be defined to store a reference to its parent (in field `p`) as well as another field `c` that represents the node's color; these fields are needed to support efficient insertions into a red-black tree while keeping the tree balanced.

5.2 Code Generation

I now describe how we generalize the technique that was presented in Section 4.2 so that it also generates efficient code to compute on sparse tensors stored in pointer-based

```
struct bst {
  pair<int32_t,double> e;
  bst* r;
  bst* l;
};
```

(a) BST

```
struct blist {
  pair<int32_t,double> e[3];
  blist* n;
  int32_t B;
};
```

(b) Block linked list

```
struct btree {
  enum type { btree_internal, btree_leaf };
  type tp;
};

struct btree_internal : public btree {
  pair<int32_t,double> e[3];
  btree* cf;
  btree* cr[3];
  int32_t B;
};

struct btree_leaf : public btree {
  pair<int32_t,double> e[3];
  int32_t B;
};
```

(c) B-tree

Figure 5-5: Examples of structs that our technique emits for storing different pointer-based data structures.

formats. As before, our technique takes as input a a tensor algebra computation expressed in concrete index notation and recursively emits imperative (C++) code to iterate (or map) over each dimension of the operand tensors. The remainder of this section will thus focus on how our technique generates code to efficiently compute on nonzeros along just one dimension. In particular, I show how specifications of pointer-based level formats that are implemented in the node schema language may be used to generate code that can be optimized in very different ways for different computations and operand formats. The result is a compiler that reduces the effort needed to efficiently work with sparse tensors stored in pointer-based formats.

5.2.1 Generating Node Type Declarations

Before generating code to compute on sparse tensors that are stored in pointer-based formats, our technique first emits code to declare structs that represent nodes of the underlying data structures and that the generated code can actually work with. These structs are directly generated from node schemas, with one struct generated for each node schema. Figure 5-5 shows representative examples of structs that our technique generates for storing some of the pointer-based data structures defined in Section 5.1.

Table 5.1 shows how our technique translates scalar fields in a node schema to fields in the corresponding struct. Array fields are translated in a similar same way, except each

Table 5.1: Translation of (scalar) fields in a node schema to fields in the corresponding emitted struct. Array fields in a node schema can be similarly translated, except with the corresponding fields in the emitted struct being arrays as well.

Field in Schema	Field in Emitted Struct	Notes
`f : elem`	`pair<int32_t,V> f`	First element of emitted pair stores coordinate of nonzero (or -1 if no nonzero is stored). Second element of emitted pair stores value of nonzero or pointer to data structure storing a subtensor.
`f : node_type`	`node_type* f`	
`f : size`	`int32_t f`	
`f : parent`	`T* f`	`T` is struct type being emitted (or its supertype, if applicable).
`f : bool`	`bool f`	
`f : [u]intN`	`[u]intN_t f`	`N` $\in \{8, 16, 32, 64\}$

emitted field is either an array member (e.g., `int32_t f[4]`) or a pointer into an array (e.g., `int32_t* f`). By default, array fields are translated to pointers to arrays that can be allocated separately from their containing struct. However, if an array field's size is either a constant `N` or upper-bounded by an `in` clause to be `N`, then the field is instead translated to an array member of size `N`; this speeds up accesses to the array at run-time by eliminating an indirection. Additionally, if a node only has one array field with an unbounded size, then our technique similarly translates the field to a zero-length array member[1] that stores its elements contiguously with the other fields of the node.

Finally, to support having different types of nodes share a common supertype, our technique emits a struct for each supertype `T`, and all other emitted structs that correspond to `T`'s subtypes inherit from `T`'s corresponding struct. `T`'s corresponding struct contains a single member `tp`, which stores an enumeration that is intended for keeping track of a node's concrete type at run-time.

5.2.2 Generating Map Functions

As pointed out in Section 4.2.3, when the non-zero elements of a tensor algebra computation's result is known to be a subset of an operand T's non-zero elements, the computation can always be performed by simply mapping over and computing with each of T's nonzeros. In particular, this is the case for all multiplicative computations such as element-wise

[1]Zero-length array members are not technically permitted in standard C++, though they are supported in practice by most C++ compilers (including GCC and LLVM for instance) as extensions.

vector multiplication ($\forall_i\ a_i = b_i c_i$), since multiplication produces a non-zero result only if all operands are also non-zero. Thus, when one operand of such a computation is stored in a pointer-based level format while the rest are stored in (array-based) formats that support efficient random access of stored elements (i.e., the locate capability), our technique emits code that maps over the pointer-based data structure to perform the computation. More precisely, our technique does this when $Coiter(e)$ (i.e., the set of operands that needs to be co-iterated in order to compute an expression e, as described in Section 4.2.3) consists of only a single operand and that operand is stored in a pointer-based level format.

Let T represent a sparse tensor stored using pointer-based level formats. To generate sequential code that maps over a dimension of T and performs some specific computation at each stored coordinate, our technique applies the algorithm shown in Figure 5-6. Our technique emits a map function for every type of node in the underlying pointer-based data structure. Figure 5-7a shows an example map function that our technique generates for mapping over nodes in a binary search tree in order to compute element-wise vector multiplication on a vector that is stored using the `bst` level format. Each emitted function processes all `elem` fields in the input node. At each coordinate stored in an `elem` field, the emitted function performs the specified computation with the corresponding nonzero (or non-empty subtensor) of T, which is also stored as part of the `elem` field (lines 6–7 in Figure 5-7a). If the computation involves other operands, then the emitted function random accesses those other operands (by invoking their implementations of `locate`) in order to load their corresponding elements before performing the computation (line 7 in Figure 5-7a). Additionally, in the general case, the emitted function maps over nonzeros that are stored in descendants of the input node by (recursively) invoking the appropriate map function to process each child of the input node (lines 4–5 and 8–9 in Figure 5-7a). By default, as shown in Figure 5-7a, our technique emits code that computes on T's stored nonzeros in coordinate order as specified by the input node's sequence attribute. However, if the input node does not provide a sequence attribute, then our technique simply emits code to process the input node's fields in the order they are declared in the node schema.

The above approach generates correct code for any pointer-based data structure that can be expressed using the node schema language. However, for data structures such as

```
emit_map(iv, expr, b):
  body := lower(expr, b)  # emit code to compute expr

  V := ...  # data type of elements in tensor b
  vars = ...  # variables referenced by body
  params := [", typeof(v) v" foreach v in vars].join()
  args := [", v" foreach v in vars].join()

  foreach schema in tensor b's node schemas:
    T := ...  # node type declared by schema
    emit       "void map_b(T* b params) {"
    emit         "if (b) {"
    if schema is a supertype:
      foreach aschema in tensor b's node schemas:
        S := ...  # node type declared by aschema
        if S is subtype of T:
          emit     "if (b->tp == T::S)"
          emit       "map_b((S*)b args);"
    else:
      foreach term in schema's sequence attribute:
        if term is a field:
          if term is an elem field:
            emit   "{"
            emit     "int32_t iv = b->term.first;"
            emit     "V v = b->term.second;"
            if term is not nonempty:
              emit   "if (iv != -1) {"
            emit       "body"
            if term is not nonempty:
              emit   "}"
            emit   "}"
          else:  # term is a child field
            if term is not nonempty:
              emit "if (b->term)"
            emit     "map_b(b->term args);"
        else:  # term is a set of array fields
          if size of array fields is constant N:
            bnd := N
          else:
            sz := ... # field storing size of arrays
            bnd := "b->sz"
          emit     "for (int32_t p = 0; p < bnd; p++) {"
          foreach entry in term:
            if entry is an elem field:
              emit   "{"
              emit     "int32_t iv = b->term[p].first;"
              emit     "V v = b->term[p].second;"
              if term is not nonempty:
                emit   "if (iv != -1) {"
              emit       "body"
              if term is not nonempty:
                emit   "}"
              emit   "}"
            else:  # entry is a child field
              if term is not nonempty:
                emit "if (b->term[p])"
              emit     "map_b(b->term[p] args);"
          emit     "}"
    emit         "}"
    emit       "}"
```

Figure 5-6: Algorithm for generating sequential code that maps over nonzeros in operand tensor b in order to compute the concrete index notation statement $\forall_{iv}\ expr$. The algorithm assumes that each node schema contains a sequence attribute; if there is not one, the algorithm emits code for each field in the order they are declared in the node schema. The algorithm also does not show tail call optimization applied.

```
1  void map_b(bst* b, double* a,
2             double* c) {
3    if (b) {
4      if (b->l)
5        map_b(b->l, a, c);
6      int32_t i = b->e.first;
7      a[i] = b->e.second * c[i];
8      if (b->r)
9        map_b(b->r, a, c);
10   }
11 }
```

(a) Sequential map over BST

```
1  void map_b(blist* b, double* a, double* c) {
2    while (b) {
3      for (int32_t p = 0; p < b->B; p++) {
4        int32_t i = b->e[p].first;
5        a[i] = b->e[p].second * c[i];
6      }
7      b = b->n;
8    }
9  }
```

(b) Sequential map over block linked list

```
1  void map_b(bst* b, double* a,
2             double* c, uint8_t d) {
3    if (b) {
4      if (d != 0) {
5        if (b->l)
6          #pragma omp task
7          map_b(b->l, a, c, d - 1);
8        if (b->r)
9          #pragma omp task
10         map_b(b->r, a, c, d - 1);
11       int32_t i = b->e.first;
12       a[i] = b->e.second * c[i];
13     } else {
14       map_b(b, a, c);
15     }
16   }
17 }
```

(c) Parallel map over BST

```
1  void map_b(blist* b, double* a, double* c) {
2    while (b) {
3      #pragma omp task
4      for (int32_t p = 0; p < b->B; p++) {
5        int32_t i = b->e[p].first;
6        a[i] = b->e[p].second * c[i];
7      }
8      b = b->n;
9    }
10 }
```

(d) Parallel map over block linked list

Figure 5-7: Examples of map functions that our technique emits. Note that Cilk-parallelized code can be similarly generated by replacing OpenMP pragmas with Cilk keywords.

linked lists that do not exhibit any fanout (i.e., those comprised of nodes that each has exactly one child), this approach can cause stack overflows at run time if the input data structure contains too many nodes. Thus, for any type of node that has exactly one child of the same type, our technique applies tail call optimization to instead emit a map function that uses a loop to iterate over all of the input node's descendants. (Our technique can trivially determine if a type of node has exactly one child of the same type by inspecting its schema.) Figure 5-7b shows an example of code that our technique emits for mapping over a block linked list using this approach.

Our technique also generates parallelized map functions in a similar way as sequential map functions. For any input node that has exactly one child of the same type (e.g., block linked list nodes), our technique parallelizes the processing of its descendants by emitting code that spawns a new task to compute on nonzeros stored in each node (lines 3–7 in

Figure 5-7d). Meanwhile, for all other types of nodes, our technique emits code that spawns new parallel tasks to process each child of the input node (lines 5–10 in Figure 5-7c). To avoid spawning too many fine-grained tasks, the emitted code keeps track of the depth of recursion (parameter `d` in Figure 5-7c) and, once a certain depth has been reached, switches back to a sequential version of the map function (lines 13–14 in Figure 5-7c).

Finally, to support mapping over nodes that are subtypes of some supertype, our technique emits a map function for each supertype that simply checks the input node's concrete type and invokes the concrete type's map function in order to actually compute on the input node. So to map over a child of a B-tree node, for instance, the generated code would invoke a map function that takes any instance of `btree` as argument. This function would, in turn, simply invoke a second map function (which performs the actual computation) that only takes an instance of either `btree_internal` or `btree_leaf` as argument, depending on if the child is an internal node (i.e., if `tp == btree::btree_internal`) or a leaf node.

5.2.3 Generating Iterators

In general though, computing a sparse tensor algebra operation may require co-iterating over multiple operands that are all stored in pointer-based level formats, which cannot be done using map functions like those described in the previous subsection. To support such computations, our technique emits code that uses a set of loops to co-iterate over the operands and compute with the intersections or unions of their nonzeros, as Figure 1-5c demonstrates for instance. A compiler can apply the same algorithm that was described in Section 4.2.5 in order to generate loops that co-iterate over operands stored in pointer-based level formats. Unlike with array-based level formats though, our technique does not require users to implement level functions that describe how to iterate over stored coordinates in pointer-based level formats. Thus, the compiler has to also automatically generate efficient iterators for operands that are stored in pointer-based level formats, given only node schemas that define those formats.

To generate an iterator that enumerates stored coordinates (and corresponding values) in a pointer-based level format, our technique first mechanically emits a (recursive) coroutine for every type of node that may be contained in the underlying data structure. This

```
emit_iterator(b):
  V := ...  # data type of elements in tensor b

  foreach schema in tensor b's node schemas:
    T := ...  # node type declared by schema
    emit       "pair<int32_t,V> iter_T(T* b) {"
    emit         "if (b) {"
    if schema is a supertype:
      foreach aschema in tensor b's node schemas:
        S := ...  # node type declared by aschema
        if S is subtype of T:
          emit     "if (b->tp == T::S)"
          emit       "yield iter_S((S*)b);"
    else:
      foreach term in schema's sequence attribute:
        if term is a field:
          if term is an elem field:
            if term is not nonempty:
              emit "{"
              emit   "int32_t i = b->term.first;"
              emit   "if (i != -1)"
            emit       "yield b->term;"
            if term is not nonempty:
              emit "}"
          else:  # term is a child field
            if term is not nonempty:
              emit "if (b->term)"
            S := ...  # node type of term
            emit     "yield iter_S(b->term);"
        else:  # term is a set of array fields
          if size of array fields is constant N:
            bnd := N
          else:
            sz := ... # field storing size of arrays
            bnd := "b->sz"
          emit     "for (int32_t p = 0; p < bnd; p++) {"
          foreach entry in term:
            if entry is an elem field:
              if term is not nonempty:
                emit "{"
                emit   "int32_t i = b->term[p].first;"
                emit   "if (i != -1)"
              emit       "yield b->term;"
              if term is not nonempty:
                emit "}"
            else:  # entry is a child field
              if term is not nonempty:
                emit "if (b->term[p])"
              S := ...  # node type of term
              emit     "yield iter_S(b->term[p]);"
          emit     "}"
    emit         "}"
    emit       "}"
```

Figure 5-8: Algorithm for generating unoptimized iterators that enumerate nonzeros of any tensor stored in the same format as tensor b.

is done by applying the algorithm shown in Figure 5-8. Figure 5-9a shows an example coroutine that our technique generates for iterating over a binary search tree. Each emitted coroutine accesses all of the input node's stored nonzeros (or, more generally, non-empty subtensors) and child nodes in the order specified by the input node's sequence attribute. For each nonzero, the emitted code simply yields the coordinate and value of that nonzero

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    if (n) {
3      if (n->l)
4        yield iter_bst(n->l);
5      yield n->e;
6      if (n->r)
7        yield iter_bst(n->r);
8    }
9  }
```

(a) Unoptimized iterator

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    while (n) {
3      if (n->l)
4        yield iter_bst(n->l);
5      yield n->e;
6      n = n->r;
7    }
8  }
```

(b) After tail call optimization

```
1  pair<int32_t,double> iter_bst(bst* n) {
2    call_stack<uint8_t,bst*> cs;
3    cs.emplace(0, n);
4    while (!cs.empty()) {
5      n = get<1>(cs.top());
6      if (get<0>(cs.top()) == 1)
7        goto call_resume1;
8      while (n) {
9        if (n->l) {
10         get<0>(cs.top()) = 1;
11         get<1>(cs.top()) = n;
12         cs.emplace(0, n->l);
13         goto call_end;
14 call_resume1:;
15       }
16       yield n->e;
17       n = n->r;
18     }
19     cs.pop();
20 call_end:;
21   }
22 }
```

(c) After recursion elimination

Figure 5-9: Steps involved in generating an optimized iterator for BSTs. The final code is shown in Figure 1-5c.

(line 5 in Figure 5-9a). For each child node, on the other hand, the emitted code (recursively) invokes the appropriate coroutine to yield all nonzeros that are stored in the child node and its descendants (lines 3–4 and 6–7 in Figure 5-9a). The same technique applies if the input node stores non-empty subtensors as opposed to (scalar) nonzeros, except the emitted coroutine yields references to the stored non-empty subtensors instead.

Our technique then applies a set of optimizations to each emitted coroutine in order to obtain a significantly more optimized iterator. First our technique applies tail call optimization in order to reduce the number of recursive calls. Additionally, if the input node has child nodes of other types, all invocations of iterators for those nodes are inlined, yielding a coroutine that only has recursive calls to itself. Then, to eliminate the overhead of recursive calls to a coroutine, our technique rewrites the coroutine so that it emulates recursion using a loop with an explicit call stack, which stores the local variables and state of each recursive call. Finally, to obtain code that does not rely on language support for coroutines (and that can thus be compiled with pre-C++20 compilers or even trivially translated to C), our technique rewrites the coroutine to a function that, when invoked, yields the next stored coordinate and value as output parameters. Figure 5-9 show how

our technique applies these optimizations to the unoptimized code in Figure 5-9a in order to generate an efficient iterator for the `bst` level format, which is shown in Figure 1-5c (i.e., the function `iter_bst`). Then, by applying a straightforward extension of the algorithm described in Section 4.2.5, a compiler can emit code that uses the generated iterator to co-iterate over tensors that are stored using BSTs and tensors that may be stored in any other format, including block linked lists (Figure 1-5c) or sparse arrays (Figure 6-3).

The iterators that our technique generates, unlike the recursive map functions described in the previous subsection, can be used for any sparse tensor algebra computation. This includes all computations that are readily supported by recursive map functions. Nevertheless, such computations can generally still be more efficiently performed with map functions that our technique generates. For one thing, our technique can often generate efficient parallelized map functions to perform these computations, whereas iterators that our technique generates are inherently sequential. For another, as I will demonstrate in Section 7.3.4, iterators that our technique generates may incur non-negligible performance overhead even when compared to sequential map functions. Thus, only for computations that cannot be readily implemented using recursive map functions (i.e., those that have to co-iterate over multiple sparse operands) does our technique default to generating code that relies on iterators.

5.3 Chapter Summary

In this chapter, we saw how pointer-based level formats can be precisely defined by specifying how the underlying data structures organize stored elements in memory. We further saw how a compiler can use such specifications of pointer-based data structures to generate efficient code for computing on sparse tensors that are stored in pointer-based level formats. In the next chapter, we will see how our technique can also generate efficient code for storing the results of sparse tensor operations in pointer-based level formats.

Chapter 6

Supporting Sparse Tensor Assembly

In the previous two chapters, we focused on how a compiler can generate efficient code to compute on sparse tensor operands that may be stored in arbitrary combinations of array-based and pointer-based formats. However, the results of sparse tensor algebra computations are often themselves sparse, and these result tensors can also be potentially stored in any of the same formats that are used to store the operands.

In this chapter, I present an abstract interface that captures the distinct ways in which sparse tensors can be efficiently assembled in order (Section 6.1) or out of order (Section 6.2) in disparate array-based and pointer-based formats. As before, our abstract interface can be implemented for each format completely independently of other formats, which makes it practical to add support for a large number of formats. I further describe how, guided by implementations of this abstract interface, a compiler can generate efficient code to store the results of sparse tensor computations in the aforementioned formats.

6.1 In-Order Assembly

A sparse tensor is assembled *in order* if nonzeros that are grouped together in the underlying data structure (e.g., stored in the same segment of an array or in the same pointer-based data structure) are always stored into the data structure consecutively. Our assembly abstract interface exposes a set of capabilities that capture the different ways in which sparse tensors stored in array-based (Section 6.1.1) or pointer-based formats (Section 6.1.2)

can be assembled in order. In this section, I describe these capabilities in more detail and discuss how a compiler can generate efficient code to assemble the results of sparse tensor computations in order by utilizing implementations of our assembly abstract interface.

6.1.1 Array-Based Formats

Array-based level formats can support one of two capabilities—*insert* and *(array-based) append*—that provide the ability to add new coordinates to coordinate hierarchy levels in order. Table 6.1 shows examples of how different level formats can implement the level functions that expose these capabilities. As shown in Figure 4-5, our code generation algorithm emits sparse tensor algebra code that invokes the aforementioned level functions to store the coordinates of result nonzeros. In particular, the algorithm emits code that adds new a coordinate to the result tensor's coordinate hierarchy representation as soon as each result nonzero is computed. Additionally, if necessary, the algorithm emits code that, at the end of co-iteration, inserts edges into the result tensor's coordinate hierarchy representation in order to associate stored coordinates with their parents. Then, as described in Section 4.2.5, our technique can specialize the emitted code to store the result tensor in a specific format by inlining the level function calls.

The insert capability inserts coordinates into a coordinate hierarchy level at any position and is exposed as three level functions:

- `insert_coord(`$\texttt{p}_k$`, `$\texttt{i}_k$`) -> void`
- `insert_init(`$\texttt{pb}_k$`, `$\texttt{pe}_k$`) -> void`
- `segment_size() ->` $\texttt{sz}_k$

The insert capability requires the locate capability to also be supported, and the level function `insert_coord` inserts a coordinate i_k into a coordinate hierarchy level at position p_k as computed by `locate`. The level function `insert_init` initializes the data structures that encode a coordinate hierarchy level between positions pb_k and pe_k, and the level function can be invoked whenever a parent coordinate is inserted in order to preallocate space for storing its child coordinates. Finally, `segment_size` returns the amount of additional space to preallocate for storing each new parent coordinate's children.

Table 6.1: Definitions of level functions that implement in-order assembly capabilities for various array-based level formats.

Level Format	Level Function Definitions	

dense	`insert_coord(p_k, i_k):`
	`  // do nothing`
	
	`insert_init(pb_k, pe_k):`
	`  // do nothing`
		`segment_size():`
		`  return N_k`

masked	`insert_coord(p_k, i_k):`
	`  mask[p_k] = 1`
	
	`insert_init(pb_k, pe_k):`
	`  // do nothing`
		`segment_size():`
		`  return N_k`

compressed	`append_coord(p_k, i_k):`
	`  crd[p_k] = i_k`
	
	`append_edges(p_{k-1}, pb_k, pe_k):`
	`  pos[p_{k-1} + 1] = pe_k - pb_k`
		`append_init(sz_{k-1}, sz_k):`
		`  for (int p_{k-1} = 0; p_{k-1} <= sz_{k-1}; ++p_{k-1}) {`
		`  pos[p_{k-1}] = 0`
		`  }`
		
		`append_finalize(sz_{k-1}, sz_k):`
		`  int cumsum = pos[0]`
		`  for (int p_{k-1} = 1; p_{k-1} <= sz_{k-1}; ++p_{k-1}) {`
		`  cumsum += pos[p_{k-1}]`
		`  pos[p_{k-1}] = cumsum`
		`  }`

singleton	`append_coord(p_k, i_k):`
	`  crd[p_k] = i_k`
	
	`append_edges(p_{k-1}, pb_k, pe_k):`
	`  // do nothing`
		`append_init(sz_{k-1}, sz_k):`
		`  // do nothing`
		
		`append_finalize(sz_{k-1}, sz_k):`
		`  // do nothing`

hashed	`insert_coord(p_k, i_k):`
	`  crd[p_k] = i_k`
	
	`insert_init(pb_k, pe_k):`
	`  for (int p_k = pb_k; p_k < pe_k; ++p_k) {`
	`  crd[p_k] = -1`
	`  }`
		`segment_size():`
		`  return W_k`

The append capability for array-based level formats appends coordinates to a coordinate hierarchy level and is also exposed as four level functions:

- `append_coord(p_k, i_k) -> void`
- `append_edges(p_{k-1}, pb_k, pe_k) -> void`
- `append_init(sz_{k-1}, sz_k) -> void`
- `append_finalize(sz_{k-1}, sz_k) -> void`

The level function `append_coord` appends a coordinate i_k to the end of a coordinate hierarchy level (at position p_k). The level function `append_edges`, meanwhile, inserts edges that connect all coordinates between positions pb_k (inclusive) and pe_k (exclusive) to the

coordinate at position p_{k-1} in the parent level. Additionally, `append_init` initializes the data structures that encode a coordinate hierarchy level, while `append_finalize` performs any post-processing that might be required after all coordinates have been append. Both level functions take, as inputs, the size of the level being initialized or finalized (sz_k) as well as the size of its parent level (sz_{k-1}). For a level that provides the insert capability, its size is computed as the size of its parent level multiplied by the amount of space allocated for storing each parent coordinate's children (i.e., the amount returned by `segment_size`). For a level that supports the append capability, on the other hand, its size is the number of coordinates that have been appended to the level. Following the semantics of the append capability, one can, for instance, assemble the `crd` array of a CSR output matrix by repeatedly calling `append_coord`—as defined for the compressed level format (see Table 6.1)—with the coordinates of every result nonzero as arguments. Similarly, the `pos` array can be assembled with calls to `append_init` at the start of the computation, `append_edges` after the nonzeros of each row have been appended, and `append_finalize` at the very end.

6.1.2 Pointer-Based Formats

Pointer-based level formats can similarly support capabilities—namely *(pointer-based) append* and *bulk assembly*—that provide the ability to store nonzeros into the underlying pointer-based data structures. The append capability for pointer-based level formats captures how nonzeros (or, more generally, non-empty subtensors) can be individually appended to the underlying data structures and is exposed as two level functions:

- `append_first(elem, st, ret) -> void`
- `append_rest(elem, st) -> void`

`append_first` exposes how the first nonzero can be appended to a pointer-based data structure, while `append_rest` exposes how all subsequent nonzeros can be appended (in order of their coordinates, if the data structure is specified by sequence attributes to be sorted). Both functions take as inputs the nonzero (or non-empty subtensor) to be appended (`elem`) as well as a reference to a user-defined object (`st`) that can be utilized to keep track of where exactly a nonzero was last appended in the data structure being

```
st = {
  node : blist
};

append_first(elem, st, ret):
  blist* node = new blist;
  node->e[0] = elem;
  node->B = 1;
  node->n = null;
  ret->h = node;
  st->node = node;

append_rest(elem, st):
  blist* node = st->node;
  if (node->B == 4) {
    node = new blist;
    node->B = 0;
    node->n = null;
    st->node->n = node;
    st->node = node;
  }
  node->e[node->B] = elem;
  node->B += 1;
```

(a) Append for `blist` level format

```
build_rbt(elems, s, e):
  if (s > e)
    return null;
  rbtree* node = new rbtree;
  uint64 m = (s + e) / 2;
  node->e = elems[m];
  node->p = null;
  node->c = (s + 1 == e);
  if (s == e) {
    node->l = node->r = null;
  } else if (s + 1 == e) {
    node->l = build_rbt(elems, s, s);
    node->r = null;
    node->l->p = node;
  } else {
    node->l = build_rbt(elems, s, m - 1);
    node->r = build_rbt(elems, m + 1, e);
    node->l->p = node;
    node->r->p = node;
  }
  return node;

build(ret, elems, sz):
  ret->r = build_rbt(elems, 0, sz - 1);
```

(b) Bulk assembly for `rbtree` level format

Figure 6-1: Examples of how the level functions that expose the pointer-based append and bulk assembly capabilities can be implemented for specific pointer-based level formats.

assembled. Additionally, `append_first` takes as input a reference to a preallocated node (`ret`) that is intended to serve as a handle to the data structure being assembled. As an example, Figure 6-1a demonstrates how the append capability can be implemented for the `blist` level format, which stores elements using block linked lists. To append the first new nonzero, the `blist` level format's implementation of `append_first` allocates a block, stores the nonzero at the beginning of the block, and initializes the root pointer (`ret->h`) to point to the block. For each subsequent new nonzero, the `blist` level format's implementation of `append_rest` then simply appends the new nonzero to the end of the last allocated block (which is cached in `st`) unless the block is already full, in which case a new block is first allocated and attached to the rest of the list.

The bulk assembly capability, on the other hand, captures how a pointer-based data structure can be constructed in one shot from a set of nonzeros (or, more generally, non-empty subtensors) and is exposed as a single level function:

- `build(elems, sz, ret) -> void`

where `elems` represents the sequence of nonzeros (or, more generally, non-empty subtensors) to be inserted, `sz` stores the size of `elems`, and `ret` is again a reference to a preallocated node that is intended to serve as a handle to the data structure being assembled. The argument `elems` implements an array interface, so any nonzero can be accessed by their position in the sequence. Additionally, if the data structure being assembled is specified to be sorted (i.e., if stored nonzeros are ordered by a sequence attribute), then the nonzeros in `elems` are guaranteed to be ordered by their coordinates. As an example, Figure 6-1b shows how the bulk assembly capability can be implemented for a level format that stores elements using red-black trees (`rbtree`). The bulk assembly capability can often be implemented more efficiently than the append capability. In the case of red-black trees, for instance, bulk assembly can be performed without needing to rebalance the tree for each inserted nonzero, which by contrast is needed when appending to red-black trees. Furthermore, bulk assembly is typically more amenable to parallelization; for example, the implementation of `build` in Figure 6-1b can be trivially parallelized by having recursive calls to `build_rbt` be spawned in parallel. However, bulk assembly requires the set of inserted nonzeros (`elems`) to be fully precomputed, which for some computations may incur additional overhead.

Similar to how our technique generates code that store the results of sparse tensor computations in array-based formats, a compiler can utilize implementations of the level functions described above in order to generate sparse tensor algebra code that store results using pointer-based data structures. Specifically, the compiler first emits code that invokes `build` or `append_first` and `append_rest` to store the result nonzeros. Then, the emitted code can be specialized to a specific type of pointer-based data structure by inlining its implementation of the aforementioned level functions. So to generate code that stores the result of a sparse tensor computation in a block linked list, for instance, the compiler first emits code like what is shown in Figure 6-3, which stores result nonzeros by invoking `append_first` and `append_rest`. The compiler can then inline implementations of `append_first` and `append_rest` for the `blist` level format (as shown in Figure 6-1a) into the emitted code, yielding code that is specialized for block linked list outputs. On the other hand, if a computation simply assigns an input tensor to the output and if the input

```
bst* map_b(bst* b, double* c) {
  if (b) {
    bst* ret = new bst;
    ret->l = NULL;
    if (b->l)
      ret->l = map_b(b->l, c);
    int32_t i = b->e.first;
    ret->e.first = i;
    ret->e.second = b->e.second * c[i];
    ret->r = NULL;
    if (b->r)
      ret->r = map_b(b->r, c);
    return ret;
  }
  return NULL;
}
```

Figure 6-2: Example emitted code that element-wise multiplies a dense vector by a sparse vector stored in a BST and that stores the result in another BST by deeply copying the sparse input vector.

```
blist_head* aret;
bool afirst = true;
uint8_t bs = iter_bst(0, ..., ib, bv);
int32_t pc = c_pos[0];
while (bs && pc < c_pos[1]) {
  int32_t ic = c_crd[pc];
  int32_t i = min(ib, ic);
  if (i == ib && i == ic) {
    double av = bv * c[pc];
    if (afirst) {
      aret = new blist_head;
      append_first({i, av}, ast, aret);
      afirst = false;
    } else {
      append_rest({i, av}, ast);
    }
  }
  if (i == ib)
    bs = iter_bst(bs, ..., ib, bv);
  pc += (i == ic);
}
```

Figure 6-3: Example emitted code that multiplies a sparse vector stored in a BST by an array-based sparse vector and that invokes `append_first` and `append_rest` to store result nonzeros. Our technique can further specialize this code for block linked list outputs by inlining implementations of the append functions for block linked lists.

is stored in an array-based format, the compiler can instead emit code that invokes the `build` function (with a reference to the input as the argument `elems`) to bulk assemble the output tensor. The compiler can then inline any format's implementation of `build` into the emitted code to obtain code that bulk assembles the output tensor in that specific format.

As an optimization though, if a computation can be performed with a map function (as described in Section 5.2.2) and if the result is stored in the same format as the input tensor being mapped over, then our technique instead emits a map function that assembles the output by essentially deeply copying the input data structure. This approach is valid since each element in the result is computed from one distinct element in the input tensor being mapped over, so the compiler can infer that the output data structure must have the same structure as the input data structure. Figure 6-2 shows an example map function that our technique generates, which computes on an input tensor that is stored using a BST and which stores the result using another BST. Such map functions can be generated in largely the same way as described in Section 5.2.2. To make a deep copy the input data structure

though, each emitted map function additionally allocates and returns a new node that is of the same type as the input node (lines 3 and 13 in Figure 6-2). This new node is initialized by copying over the coordinates of all elements that are stored in the input node (line 8 in Figure 6-2), with the corresponding values initialized to be the results of the computation (line 9 in Figure 6-2). Furthermore, new output child nodes are allocated by invoking the augmented map function(s) on the input node's children (lines 6 and 12 in Figure 6-2).

6.2 Out-of-Order Assembly

While many sparse tensor computations can assemble their outputs in order, there also exist many other examples of sparse tensor operations that must instead assemble their outputs *out of order* (i.e., with nonzeros stored into the output data structures in arbitrary orders). An important representative class of such operations is sparse tensor format conversion, which can be viewed as an operation that simply takes a tensor stored in some source format and assigns it to an output tensor in a target format that typically stores nonzeros in a different order. Additionally, parallelized sparse tensor algebra computations generally do not produce result nonzeros in any fixed order, and so these result nonzeros would have to be inserted into the output tensors out of order. Unfortunately, out-of-order assembly of a sparse tensor is particularly challenging when the tensor needs to be stored in an array-based format. This is because, as shown in Figures 2-1, 2-2, and 2-3, array-based formats typically store nonzeros compactly in memory. The result is that naively inserting nonzeros one by one can incur significant performance overhead by requiring most already-stored nonzeros to be moved around in memory after each insertion.[1]

In this section, I describe how a compiler can generate efficient code to assemble the outputs of sparse tensor operations out of order, with those outputs potentially being stored in disparate array-based formats. In particular, I show how out-of-order assembly of sparse tensors in a wide range of array-based formats can be decomposed into three logical phases. I describe how our technique can generate efficient code for each of the first

[1]By contrast, out-of-order assembly of sparse tensors in pointer-based formats does not suffer from this problem (or at least experiences it to a much lesser degree), since nonzeros do not have to be stored contiguously in memory.

two logical phases. I then describe another capability exposed by our assembly abstract interface that captures how, by utilizing the outputs of the first two logical phases, nonzeros can be efficiently inserted out of order into array-based level formats. Finally, I discuss how a compiler can use implementations of this capability for disparate level formats to generate efficient code for the third logical phase of out-of-order assembly.

6.2.1 Overview

Figure 6-4 shows three examples of sparse tensor conversion routines that convert tensors between different combinations of source and target formats. As we saw in previous chapters, different sparse tensor formats often store nonzeros in different orderings, and so efficiently converting tensors between different formats often requires assembling the output tensor out of order. As the examples illustrate, efficiently assembling a sparse tensor out of order in different formats can require vastly dissimilar code. It turns out, however, that efficient algorithms for assembling a sparse tensor out of order in a wide range of disparate formats can all be decomposed into (up to) three logical phases: *remapping*, *analysis*, and *assembly*. Figure 6-4 highlights these phases in different colors.

The remapping phase computes additional coordinates for each nonzero to be stored as functions of its original coordinates. These additional coordinates can then be used in the other phases to orchestrate the assembly of tensors in structured sparse tensor formats like DIA and ELL. What additional coordinates are computed depends on the output format. For instance, the code in Figure 6-4a, which converts a matrix to DIA, computes a new coordinate `k` for each nonzero as the difference between its column coordinate and its row coordinate (lines 2–6 and 20–24). By contrast, the code in Figure 6-4b, which converts a matrix to ELL, assigns a distinct value to the new coordinate `k` for every nonzero that belongs in the same row of the matrix.

The analysis phase computes statistics about the tensor being assembled, which are later used to determine the amount of memory to pre-allocate for storing nonzeros in the output format. The exact statistics that are computed also depend on the output format. Figure 6-4a, for instance, computes the set of all non-empty diagonals in the matrix being assembled (lines 1–8), with distinct diagonals identified by offsets (`k`) computed in the

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

(a) CSR to DIA

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

(b) CSR to ELL

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

(c) (Unsorted) COO to CSR

Figure 6-4: Code (in C++) that convert sparse tensors between different combinations of formats. The background colors identify distinct logical phases of out-of-order output tensor assembly (green for remapping, yellow for analysis, and blue for assembly).

remapping phase. By contrast, Figure 6-4b computes the maximum number of nonzeros in any row of the matrix being assembled (lines 1–5), while Figure 6-4c computes the number of nonzeros in each row of the matrix (lines 1–6).

Finally, the assembly phase iterates over the tensor to be assembled and inserts each nonzero into the output data structures. Where each nonzero is inserted (`pB2`) again depends on the output format. Figure 6-4a computes `pB2` as a function of each nonzero's row coordinate and its offset `k` (as computed in the remapping phase), in such a way that nonzeros with the same offset are grouped together in the output (lines 25–26). By contrast, Figure 6-4c simply appends each nonzero to its row's corresponding segment in the `crd` array (line 19), with the segments being pre-allocated based on the statistics that were computed in the analysis phase (lines 8–11).

In the next three subsections, I describe how our technique generates efficient code to perform each of the three logical phases described above. While each phase is logically distinct, our technique can emit code that fuses different phases if it is beneficial to do so. This enables a compiler to generate code like Figure 6-4a, for instance, which duplicates and fuses remapping with the analysis and assembly phases in order to avoid materializing the offsets of nonzeros. At the same time, for assembling sparse tensors in formats that store nonzeros in more complex orderings (e.g., Morton order), the compiler can emit code to perform remapping separately and materialize the additional coordinates. This eliminates the need to recompute complex remappings.

6.2.2 Remapping Phase

As we saw in Section 3.2, any structured sparse tensor format can be cast as a format for storing remapped tensors obtained by applying a coordinate remapping that captures how the format groups together stored nonzeros in memory. So to assemble tensors in any particular structured sparse tensor format, our technique emits code that, as it enumerates the nonzeros to be stored, computes additional coordinates as functions of each nonzero's original coordinates by applying the output format's coordinate remapping. This, as we will see in the next two subsections, enables our technique to—in a dimension-by-dimension

fashion—generate code that assembles the output of a sparse tensor operation in any arbitrary format, including structured sparse tensor formats.

To compute additional coordinates that are defined purely as arithmetic or bitwise expressions of a nonzero's original coordinates, our technique simply inlines those expressions directly into the emitted code (e.g., lines 6 and 24 in Figure 6-4a, which compute the first coordinate in the output of the remapping `(i,j) -> (j-i,i,j)`). Remappings that contain let expressions are lowered by first emitting code to initialize the local variables and then inlining the expressions that use those local variables. For example, a remapped coordinate `r=i/N in (r&1)|((r&2)<<2)` would be lowered to

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

Coordinate remappings that contain counters are lowered by emitting a counter array for each distinct counter in the remapping. Each element in the counter array corresponds to a distinct set of coordinates $(i_1, \ldots, i_k)$ that can be used to index into the counter, and the counter array element tracks how many input nonzeros with coordinates $(i_1, \ldots, i_k)$ have been iterated over so far. Our technique additionally emits code that, for each nonzero having coordinates that correspond to counter array element c, first assigns the nonzero to the slice of the remapped tensor that is indexed by c and then increments c. So to apply the remapping `(i,j) -> (#i,i,j)` to nonzeros that are originally stored in an unsorted COO matrix, for instance, our technique emits the following code:

```
int counter[N] = {0};  // counter array for #i
for (int p = pos[0]; p < pos[1]; p++) {
  int i = A1_crd[p];
  int j = A2_crd[p];
  int k = counter[i]++;  // k == #i
  // map A(i,j) to coordinates (k,i,j) ...
```

If the coordinates used to index into a counter are enumerated in sequence though, our technique reduces the size of the counter array in the generated code by having the counter array be reused across iterations of loops that enumerate the coordinates. Specifically, assume the sparse tensor operation being compiled is expressed in concrete index notation in the form $\forall_{i_1} \cdots \forall_{i_p} \cdots \forall_{i_n}$ *expr*, where the coordinates $i_1, \ldots, i_p$ are all used to index into

a counter. If the operands are stored using level formats that possess the ordered (or unique) property, then our technique emits a counter array that does not need to be indexed by the coordinates $i_1, \ldots, i_p$ and that can be shared across iterations of loops iterating over those same coordinates. So, for instance, to apply the same coordinate remapping as before to nonzeros that are originally stored in a CSR matrix, our technique emits optimized code as shown on lines 8–13 in Figure 6-4b, which uses the same scalar `count` variable to remap nonzeros in every row. Such an optimization can significantly improve performance by reducing memory traffic at run time.

6.2.3 Analysis Phase

As we also saw in Section 6.2.1, to avoid having to constantly reallocate and shuffle around already-stored nonzeros, code that efficiently assembles the output of a sparse tensor operation out of order typically allocates memory in one shot based on some statistics about the output tensor. Efficiently computing these statistics, however, can require very different code depending on the sparse tensor operation itself. For instance, code to convert a matrix to ELL (without dynamically resizing the `crd` and `vals` arrays) must first determine the maximum number of nonzeros K stored in any row. If the matrix is originally stored in COO, then computing K requires constructing a histogram that records the number of nonzeros in each row, which in turn requires examining all the nonzeros in the matrix. On the other hand, if the matrix is originally stored in CSR, then the number of nonzeros in each row can instead be directly computed from the `pos` array. Optimized code for converting a CSR matrix to ELL thus does not need to make multiple passes over the matrix's nonzeros, thereby reducing memory traffic.

We develop a new language called the *attribute query language*, which describes statistics of sparse tensors as aggregations over the coordinates of their nonzeros. The attribute query language is declarative, and attribute queries are specified independently of how a tensor is actually stored. This lets our technique lower attribute queries to equivalent sparse tensor computations and then simply leverage the techniques described in the previous two chapters in order to generate optimized code for computing tensor statistics. As we will see in Section 6.2.4, our technique can thus generate efficient code to perform

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

j	nic
0	3
1	2
2	0
3	1
4	1
5	0

```
select [j] ->
  min(i) as minic,
  max(i) as maxic
```

j	minic	maxic
0	0	3
1	0	1
2	-1	4
3	3	3
4	3	3
5	-1	4

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

j	nec
0	1
1	1
2	0
3	1
4	1
5	0

Figure 6-5: Examples of attribute queries computed on the tensor shown in Figure 2-2a.

any sparse tensor computation while only requiring users to provide simple-to-specify attribute queries for each potential output format, as opposed to complicated loop nests for every possible computation.

6.2.3.1 Attribute Query Language

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

`select [`$\mathtt{i}_1,\ldots,\mathtt{i}_m$`] -> <`$\mathtt{aggr}_1$`> as label`$_1$`, ..., <`$\mathtt{aggr}_n$`> as label`$_n$

where each $\mathtt{i}_k$ is a coordinate into dimension I_k of some r-dimensional tensor A and <$\mathtt{aggr}_k$> invokes the aggregation function `count`, `max`, `min`, or `id`. The result of an attribute query is conceptually a map that, for every distinct set of coordinates $(i_1, \ldots, i_m)$, stores computed statistics about the $I_{m+1} \times \cdots \times I_r$ subtensor A' in A that those coordinates uniquely identify. Figure 6-5 shows three examples of attribute queries computed on the same tensor.

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

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

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

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

computes the number of nonzeros in each $J \times K$ slice. Figure 6-5 (left) shows how `count` queries can be used to compute the number of nonzeros in each column of a matrix, which is required when assembling a sparse matrix out of order in CSC for instance.

The `max(i`$_{m+1}$`)` and `min(i`$_{m+1}$`)` functions compute, for each subtensor A', the largest and smallest coordinates i_{m+1} such that the i_{m+1}-th slice of A' along dimension I_{m+1} contains nonzeros. For instance, if `Q` denotes the result of the query in Figure 6-5 (middle), then `Q[1].minic == 0` and `Q[1].maxic == 1` since all nonzeros in column 1 of the tensor in Figure 2-2a lie between rows 0 and 1. As the example in Figure 6-5 (middle) also demonstrates, `max` and `min` return results that are outside of the domain of valid coordinates (e.g., for columns 2 and 5) if a subtensor is empty.

Finally, the `id` function simply returns 1 if a subtensor A' contains nonzeros and 0 otherwise. So if `R` denotes the result of the query in Figure 6-5 (right), then `R[1].nec == 1` since column 1 contains a nonzero while `R[2].ne == 0` since column 2 is empty.

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

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

on A results in a bit set that encodes the set of all nonzero diagonals in B. This, as mentioned in Section 6.2.1, is precisely the information required to assemble B out of order in DIA. Furthermore, since the coordinate of each slice of A is defined to be the offset of the corresponding diagonal in B from the main diagonal, applying the query

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

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

6.2.3.2 Code Generation

To generate efficient code that computes an attribute query, our technique reformulates the query as sparse tensor algebra computation. The query is first lowered to a canonical form in concrete index notation, which we extend with the ability to index into the output

tensor using coordinates that are computed as arbitrary functions of index variables. The canonical form of the query is subsequently optimized by applying a set of predefined transformations to simplify the computation. Finally, the optimized query in concrete index notation is compiled to imperative code by directly leveraging the techniques that were described in the previous two chapters. This approach works as long as the results of attribute queries are stored in a format, such as dense arrays, that can itself be efficiently assembled without needing attribute queries.

More precisely, let A be an $I_1 \times \cdots \times I_r$ tensor obtained by applying some remapping to a $J_1 \times \cdots \times J_n$ tensor B. (To simplify the explanation of our technique, we may assume that the elements of B are physically stored in memory. However, our technique does not rely on this assumption and works even if B is the result of some arbitrary sparse tensor algebra computation, as long as the computation enumerates each nonzero of B at most once and does not scatter into B.) Then, to compute an attribute query of the form

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

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

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

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

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

```
select [i_1,...,i_m] -> count(i_{m+1},...,i_l) as Q
```

on A to their canonical form

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

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

Our technique also generates code for `max` queries

```
select [i_1,...,i_m] -> max(i_{m+1}) as Q
```

by lowering them to their canonical form

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

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

```
select [i_1,...,i_m] -> min(i_{m+1}) as Q
```

are lowered to their canonical form

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

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

After an attribute query is lowered to its canonical form, our technique eagerly applies a set of predefined transformations on the query computation. Table 6.2 describes trans-

Table 6.2: Transformations that our technique applies to optimize attribute queries.

Transformation	Definition	Preconditions/Postconditions
reduction-to-assign	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \oplus\!= expr)$ $\implies (\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} = expr)$	For each j_k, there exists an i_l such that $j_k \equiv i_l$. $\oplus$ is any reduction operator. A is initialized to the zero tensor.
inline-temporary	$(\forall_{i_1} \cdots \forall_{i_m} A_{i_1 \cdots i_l} \oplus\!= f(W_{i_1 \cdots i_m}))$ **where** $(\forall_{j_1} \cdots \forall_{j_n} W_{i_1 \cdots i_m} = expr)$ $\implies (\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_l} \oplus\!= f(expr))$	f is any function that takes only W as tensor operand. $\oplus$ is any reduction operator or a simple assignment.
simplify-width-count	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \mathrel{+}= \text{map}(B_{j_1 \cdots j_n}, c))$ $\implies (\forall_{j_1} \cdots \forall_{j_{n-1}} A_{i_1 \cdots i_m} \mathrel{+}= B'_{j_1 \cdots j_{n-1}} \cdot c)$	B possesses the zeroless property, and j_n is a reduction variable that indexes into the innermost dimension of B (i.e., J_n). c is any constant. B' is a tensor that encodes the number of nonzeros in each slice of B indexed by coordinates $(j_1, ..., j_{n-1})$; values of B' are not materialized but computed on-the-fly with calls to `coord_bounds` or `pos_bounds` for dimension J_n of B.
counter-to-histogram	$(\forall_{j_1} \cdots \forall_{j_n} A_{i_1 \cdots i_m} \text{ max}\!= \text{map}(expr, \#j_k \cdots j_l + 1))$ $\implies (\forall_{i_1} \cdots \forall_{i_l} A_{i_1 \cdots i_m} \text{ max}\!= W_{i_1 \cdots i_l})$ **where** $(\forall_{j_1} \cdots \forall_{j_n} W_{i_1 \cdots i_m j_k \cdots j_l} \mathrel{+}= \text{map}(expr, 1))$	

formations that our technique uses. These transformations can optimize the performance of query computations by, for instance, reducing the number of dimensions that have to be iterated and eliminating redundant temporaries.

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

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

Our technique then proceeds to iteratively and eagerly apply the transformations shown in Table 6.2 on the computation above. In particular, each iteration variable bound to a $\forall$ is used to independently index into a dimension of W, so the substatement that defines W satisfies the preconditions of the reduction-to-assign transformation. Our technique thus applies the aforementioned transformation on the substatement that defines W to obtain

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

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

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

which is then trivially rewritten to $\forall_i \forall_j \; Q_i \mathrel{+}= \text{map}(B_{ij}, 1)$ by applying constant folding. At this point, if B is stored in memory in the (unsorted) COO format, then we can directly apply the technique described in Chapter 4 to compile this rewritten statement to imperative code that is shown on lines 1–6 in Figure 6-4c. However, if B is stored in CSR without explicit zeros, then our technique can additionally apply the simplify-width-count transformation (followed by the reduction-to-assign transformation again) to get the final query

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

where each element of B' is dynamically computed as `pos[i+1] - pos[i]`. This optimized query avoids iterating over B's nonzeros, thereby reducing memory traffic at run time.

6.2.4 Assembly Phase

To conclude this section, I first show how our assembly abstract interface exposes out-of-order assembly as a capability that utilizes the results of attribute queries to efficiently insert (remapped) nonzeros into array-based formats in any arbitrary order. I then discuss how a compiler can utilize implementations of this capability to generate efficient code that assembles the results of arbitrary sparse tensor operations out of order, with output tensors potentially being stored in disparate array-based formats.

6.2.4.1 Out-of-Order Assembly Capability

The out-of-order assembly capability captures how coordinates of nonzeros can be inserted out of order into a coordinate hierarchy level, with each level being assembled assuming all previous levels in the coordinate hierarchy have already been assembled. The assembly of each level is decomposed into (up to) two logical subphases: *edge insertion* and *coordinate*

insertion. Each subphase is defined in terms of a set of level functions, and Figure 6-6 shows how these level functions can be implemented for different array-based level formats.

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

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

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

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

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

which all level formats must also implement. `unseq_init_edges` initializes data structures that a level format uses to logically store edges. Then, for each position p_{k-1} (which represents a subtensor with coordinates $(i_1, \ldots, i_{k-1})$) in the parent level, `unseq_insert_edges` allocates some number of child coordinates to be connected to the coordinate at p_{k-1}. The number of child coordinates allocated can be computed as any arbitrary function of q_k. Finally, `unseq_finalize_edges` inserts edges into the coordinate hierarchy so that each coordinate in the parent level is connected to as many children as it was previously allocated. Figure 6-7a shows how these level functions can logically be invoked to bulk insert edges into a coordinate hierarchy.

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

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

```
init_coords(sz_{k-1}, Q_k):
  W = Q_k[0][].w + 1;

get_pos(p_{k-1}, i_1, ..., i_k):
  return p_{k-1} * W + i_k;

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

Q_k := [select [] -> max(i_k) as w]
```

(a) `sliced` level format

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

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

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

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

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

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

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

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

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

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

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

(b) `compressed` level format

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

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

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

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

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

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

(c) `squeezed` level format

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

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

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

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

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

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

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

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

(d) `ragged` level format

Figure 6-6: Examples of how the out-of-order assembly capability, including level function definitions and the required attribute queries, can be implemented for four level formats.

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

(a) Unsequenced edge insertion

```
sz_{k-1} = get_size_{k-1}(get_size_{k-2}(···(1)···));
seq_init_edges(sz_{k-1}, Q_k);
for (position p_{k-1} in parent level |
      ∃ coordinates i_1, ..., i_{k-2} connecting
      coordinate at p_{k-1} to root) {
  q_k[:] = Q_k[:][i_1, ..., i_{k-1}];
  seq_insert_edges(p_{k-1}, i_1, ..., i_{k-1}, q_k);
}
```

(b) Sequenced edge insertion

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

(c) Coordinate insertion

Figure 6-7: Out-of-order assembly capability expressed in terms of calls of level functions.

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

The coordinate insertion subphase logically inserts coordinates of nonzeros into a coordinate hierarchy. This subphase models the assembly of data structures that store the coordinates and values of nonzeros, and it is defined in terms of five level functions:

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

`init_coords` allocates and initializes data structures for storing coordinates in a coordinate hierarchy level. If a level format implicitly encodes coordinates (e.g., as a fixed range) using some fixed set of parameters, then `init_coords` also compute those parameters as

functions of the attribute query results Q_k. On the other hand, if a level format explicitly stores coordinates in memory, then the coordinates of nonzeros are inserted by invoking `insert_coord` for each nonzero. The position p_k at which each nonzero should be inserted is computed by invoking either `get_pos` or `yield_pos`. The former guarantees that nonzeros with the same coordinates are inserted at the same position. The latter allows duplicate coordinates to be inserted at different positions. Both functions, however, may rely on auxiliary data structures to track where to insert coordinates; `init_{get|yield}_pos` and `finalize_{get|yield}_pos` initializes and cleans up those data structures. Figure 6-7c shows how these level functions can logically be invoked to perform coordinate insertion.

6.2.4.2 Code Generation

Again, similar to how our technique generates code that assembles the results of sparse tensor operations in order, a compiler can utilize implementations of the level functions described above in order to emit code that assembles the results of sparse tensor operations out of order. Specifically, a compiler can use the techniques described in the previous two chapters and in Section 6.2.2 to emit loops (or map functions) that perform the desired sparse tensor operation and that apply the output format's coordinate remapping to each result nonzero. Then, within each loop nest (or map function) that enumerates the result nonzeros, the compiler can emit code that invokes the level functions described in Section 6.2.4.1 to store each nonzero into the coordinate hierarchy representation of the output tensor. The emitted code is finally specialized to the desired output format by inlining its implementations of the aforementioned level functions (e.g., as shown in Figure 6-1). Again, this approach enables the compiler to support a wide range of disparate output tensor formats without having to hard-code support for any of those formats.

While the out-of-order assembly capability assumes that levels in a coordinate hierarchy are assembled separately, in order to minimize redundant computation and memory traffic at run time, our technique emits code that fuses the assembly of adjacent levels in the output coordinate hierarchy wherever possible. Adjacent coordinate hierarchy levels can always be assembled together as long as only the parent level requires a separate edge insertion phase (or if none do). As an example, none of the level formats that compose to

express the DIA format requires edge insertion. Thus, our technique can emit code to store the result of any sparse tensor operation in DIA by enumerating the result nonzeros just once (excluding the attribute query computation) and assembling all levels of the output's coordinate hierarchy representation (i.e., all dimensions of the output matrix) together.

For each set of output coordinate hierarchy levels (i.e., output tensor dimensions) that can be assembled together, our technique then simply has to emit code like what is shown in Figure 6-7 to perform edge insertion (if required) followed by coordinate insertion. If an output level format implements both variants of edge insertion, then our technique selects one based on whether the parent level format can be iterated in order (i.e., if it possesses the ordered property or if it supports coordinate value iteration). If an output level format implements `yield_pos` but does not permit storing duplicates of the same coordinate (i.e., if it possesses the unique property), a compiler would also need to emit logic to perform deduplication on the fly by keeping track of inserted coordinates.

To see how our technique works more concretely, suppose we are generating code to convert COO matrices to CSR. To obtain code that assembles the column dimension of the output, the compiler first emits sequenced edge insertion code that has the same structure as Figure 6-7b. The emitted code is then specialized to CSR by replacing the level function calls with the `compressed` level format's implementations of those functions (Figure 6-6b). The result is lines 7–11 in Figure 6-4c, which reserves memory for each row to store its nonzeros. In a similar way, the compiler emits coordinate insertion code that has the same structure as Figure 6-7c and then specializes it to CSR, yielding lines 12–25 in Figure 6-4c.

6.3 Chapter Summary

In this chapter, we saw how a compiler can generate efficient code to store the results of sparse tensor computations in disparate array-based and pointer-based sparse tensor formats. This can be done either in order or out of order depending on the computation. We evaluate code that our technique generates in the next chapter. In particular, as we will see, code generated by our technique has performance comparable to, if not better than, equivalent hand-optimized code implemented in existing libraries and frameworks.

Chapter 7

Evaluation

To evaluate our technique, we implemented it as extensions to the TACO sparse tensor algebra compiler. In this chapter, I show that our technique supports a wide range of sparse tensor computations, and I show that code generated by our technique has performance comparable to, if not better than, equivalent hand-optimized code implemented in existing libraries and frameworks. Additionally, for computations that are not directly implemented in existing libraries and that can only be expressed as compositions of multiple kernels, our technique typically generates code that significantly outperform the libraries. I further show that our technique supports all these computations for inputs and outputs that may be stored in a wide variety of formats, and I show how using formats that are optimized for the application and the input data can improve run-time performance.

7.1 Computing with Array-Based Formats

In this section, I demonstrate our technique's ability to generate efficient code for computing on sparse tensors that are stored in array-based formats.

7.1.1 Experiment Setup

We evaluated code that our technique generates against five hand-optimized sparse linear and tensor algebra libraries: Intel MKL [44], SciPy [47], MTL4 [38], the MATLAB Tensor

Toolbox [10], and TensorFlow [1]. Intel MKL is a C and Fortran math processing library that is heavily optimized for Intel processors. SciPy is a popular Python scientific computing library. MTL4 is a C++ library that specializes linear algebra operations for fast execution using template metaprogramming. The Tensor Toolbox is a MATLAB library that implements many kernels and factorization algorithms for dense and sparse tensors with any numbers of dimensions. TensorFlow is a machine learning library that supports some basic sparse tensor operations (in addition to dense tensor operations).

All experiments were run on a two-socket, 2.4 GHz Intel Xeon E5-2695 v2 machine with 30 MB of L3 cache per socket and 128 GB of main memory. We compiled all C++ benchmarks using GCC 5.4.0 with `-O3 -march=native -mtune=native -ffast-math` optimizations enabled, and we ran all MATLAB benchmarks using MATLAB 2016b. We ran each experiment between 10 (for longer-running benchmarks) to 100 times (for shorter-running benchmarks) under cold cache conditions and report average serial execution times. We restrict execution to a single socket using `numactl`.

We ran our experiments with real-world and synthetic tensors of varying sizes and sparsity structures as input, inspired by similar collections of test matrices and tensors from related works [13, 102]. Table 7.1 describes these tensors in more detail. The real-world tensors come from applications in many disparate domains and were obtained from the SuiteSparse Matrix Collection [30] and the FROSTT Tensor Collection [101]. We stored tensor coordinates as integers and component values as double-precision floats, except for the Tensor Toolbox's TTM and INNERPROD kernels. Those two kernels do not support integer coordinates, so we evaluate them with double-precision floating-point coordinates.

7.1.2 Sparse Matrix Computations

Our technique generates efficient sparse tensor algebra code that are specialized to the layouts and attributes (e.g., sortedness) of the operands. To demonstrate this, we first evaluated code that our technique generates for computing operations on COO, CSR, DIA, and ELL matrices against equivalent code in MKL, SciPy, MTL4, and TensorFlow.

Figure 7-1 shows performance results for sparse matrix-vector multiplication (SpMV), an important operation in many iterative methods for solving large-scale linear systems

Table 7.1: Statistics about tensors used to evaluate code that our technique generates for computing on operands stored in array-based formats.

Tensor	Domain	Dimensions	NNZ	Density	Diagonals
pdb1HYS	Protein database	36K × 36K	4,344,765	3×10^{-3}	25,577
jnlbrng1	Optimization	40K × 40K	199,200	1×10^{-4}	5
obstclae	Optimization	40K × 40K	197,608	1×10^{-4}	5
chem	Chemical master equation	40K × 40K	201,201	1×10^{-4}	5
rma10	3D CFD	46K × 46K	2,329,092	1×10^{-3}	17,367
dixmaanl	Optimization	60K × 60K	299,998	8×10^{-5}	7
cant	FEM/Cantilever	62K × 62K	4,007,383	1×10^{-3}	99
consph	FEM/Spheres	83K × 83K	6,010,480	9×10^{-4}	13,497
denormal	Counter-example problem	89K × 89K	1,156,224	1×10^{-4}	13
Baumann	Chemical master equation	112K × 112K	748,331	6×10^{-5}	7
cop20k_A	FEM/Accelerator	121K × 121K	2,624,331	2×10^{-4}	221,205
shipsec1	FEM	141K × 141K	3,568,176	2×10^{-4}	10,475
scircuit	Circuit	171K × 171K	958,936	3×10^{-5}	159,419
mac_econ	Economics	207K × 207K	1,273,389	9×10^{-5}	511
pwtk	Wind tunnel	218K × 218K	11,524,432	2×10^{-4}	19,929
Lin	Structural problem	256K × 256K	1,766,400	3×10^{-5}	7
synth1	Synthetic matrix	500K × 500K	1,999,996	8×10^{-6}	4
synth2	Synthetic matrix	1M × 1M	1,999,999	2×10^{-6}	2
ecology1	Animal movement	1M × 1M	4,996,000	5×10^{-6}	5
webbase	Web connectivity	1M × 1M	3,105,536	3×10^{-6}	564,259
atmosmodd	Atmospheric model	1.3M × 1.3M	8,814,880	5×10^{-6}	7
Facebook	Social media	1.6K × 64K × 64K	737,934	1×10^{-7}	
NELL-2	Machine learning	12K × 9.2K × 29K	76,879,419	2×10^{-5}	
NELL-1	Machine learning	2.9M × 2.1M × 25M	143,599,552	9×10^{-13}	

from scientific and engineering applications [13]. Our technique is the only one that supports all the formats we survey; MTL4 does not support DIA, neither MKL nor SciPy supports ELL, and TensorFlow only supports COO. MKL and SciPy also only support the struct-of-arrays (SoA) variant of the COO format, which is the variant shown in Figure 2-2b. TensorFlow, on the other hand, only supports array-of-structs (AoS) COO, which uses a single `crd` array to store the full coordinates of each nonzero contiguously in memory. By contrast, our technique supports both variants; as mentioned in Section 3.1, supporting AoS COO only requires defining variants of the `compressed` and `singleton` level formats.

Figures 7-2 and 7-3 show performance results for sparse matrix-dense matrix multiplication (SpMM), another important operation in many data analytics and machine learning applications [56], and sparse matrix addition with COO matrices. Meanwhile, Figure 7-4 shows results for CSR matrix addition. Our technique is again the only one that supports all operations. In particular, SciPy does not support COO SpMM, while MTL4 only sup-

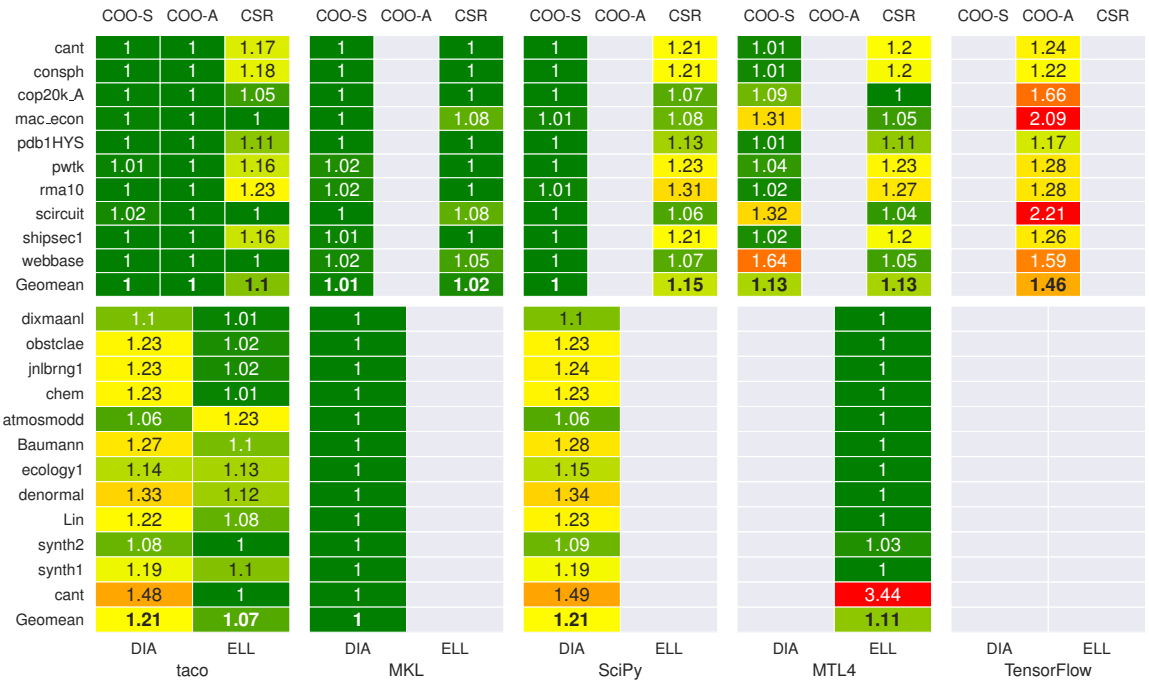


Figure 7-1: Normalized execution times of SpMV ($y = Ax$) with matrix A stored in disparate array-based formats, using code that our technique generates (taco) and that are implemented in other existing libraries. Results are normalized to that of the fastest library for each matrix and format, and the geometric means of the results are shown in bold. Unlabeled cells in gray indicate a library does not support that format; our technique the only library that supports SpMV for all the formats we evaluate. COO-S and COO-A denote the struct-of-arrays and array-of-structs variants of COO respectively.

ports SpMM with sorted COO matrices. Furthermore, only TensorFlow and our technique support computing COO matrix addition with a COO output, and TensorFlow does not support CSR matrix addition. These omissions highlight the advantage of a compiler approach such as ours that does not require every kernel to be manually implemented.

Overall, the results show that our technique generates code that has performance competitive with existing libraries. For SpMM and sparse matrix addition, code that our technique generates consistently has performance equal to or better than other libraries. For SpMV, code that our technique generates outperform TensorFlow, perform similar to SciPy and MTL4, and are competitive with MKL on the whole. Even for DIA, code that our technique generates is only about 21% slower than MKL on average. These results are explained in more depth below.

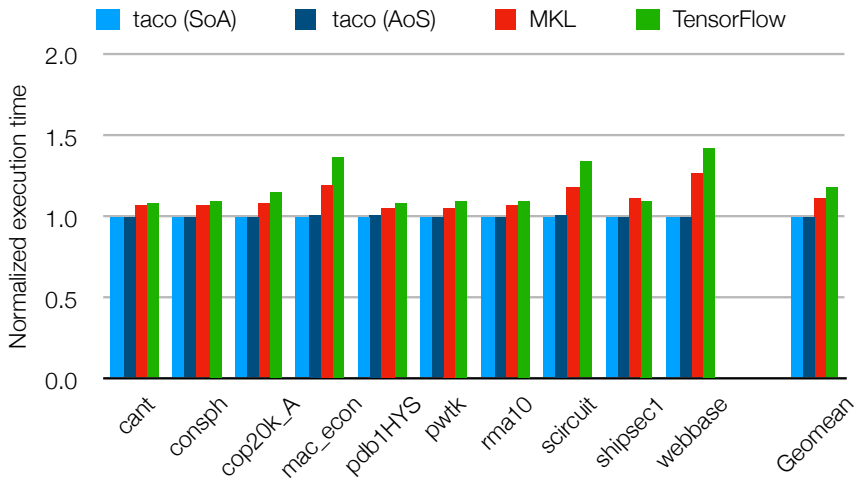


Figure 7-2: Normalized execution times of COO SpMM ($A = BC$, where B is in COO and A and C are dense matrices) with code that our technique generates (taco) and with other libraries that support the operation. Results are normalized to that of our technique (with SoA COO) for each matrix.

COO Kernels

Code that our technique generates to compute COO SpMV implements the same algorithm as SciPy and MKL, and they therefore have the same performance. MTL4 also implements this algorithm but stores the result of the computation in a temporary that is subsequently copied to the output. This incurs additional cache misses for matrices with larger dimensions, such as webbase. TensorFlow, on the other hand, does not implement a specialized COO SpMV kernel and the operation must be cast as SpMM with a single-column matrix. It therefore incurs some performance overhead because every input vector access requires a loop over its trivial column dimension.

COO matrix addition code that our technique generates is specialized to the number of dimensions that the operands possess. By contrast, TensorFlow has loops that iterate over the coordinates of each nonzero. These loops let TensorFlow support tensors with any number of dimensions using a single sparse tensor addition kernel, but again the loops introduce some overhead. TensorFlow's sparse tensor addition kernel is also hard-coded to compute with 64-bit coordinates, whereas our technique can generate code that work with narrower-width coordinates, which reduces memory traffic.

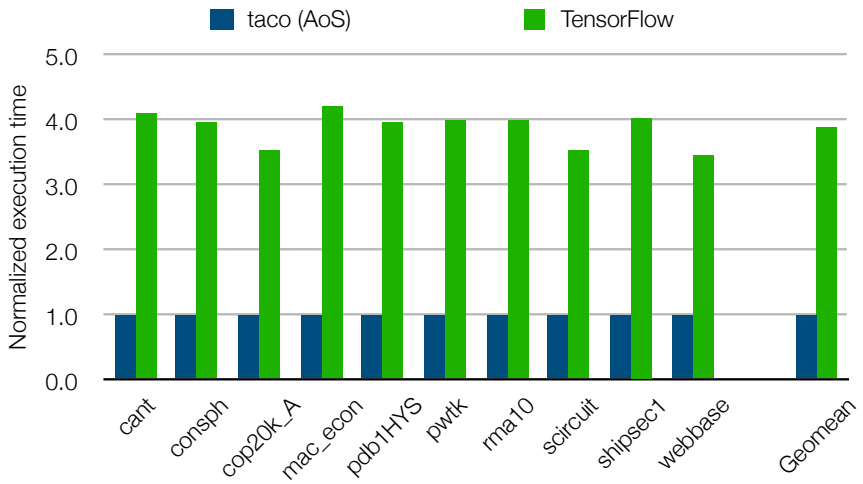


Figure 7-3: Normalized execution times of COO matrix addition ($A = B + C$, where all matrices are stored in the AoS COO format) with code that our technique generate (taco) and with TensorFlow. Results are normalized to that of our technique for each matrix.

CSR Kernels

Code that our technique generates for CSR SpMV iterates over the rows of the input matrix and, for each row, computes its dot product with the input vector. SciPy and MTL4 implement the same algorithm and thus have similar performance as code that our technique generates, while MKL also vectorizes the dot product. This, however, requires vectorizing the input vector gathers, which many SIMD architectures cannot handle very efficiently. Thus, while SIMD vectorization is beneficial for many of our test matrices (e.g., rma10), it is not always so (e.g., mac_econ).

Code that our technique generates for CSR matrix addition also uses the same algorithm as SciPy and MKL and thus has similar performance. The generated code exploits the sortedness of the input matrices to enumerate result nonzeros in order. This lets it cheaply assemble the output `crd` array by appending nonzeros to it. By contrast, MTL4 first assigns one operand to a sparse temporary and then increments it by the other operand. The latter step can require significant data shuffling to keep nonzeros stored in coordinate order within the sparse temporary, since MTL4 does not reserve memory in advance to store nonzeros. Finally, converting the temporary back to CSR incurs yet more overhead, resulting in MTL4's poor performance.

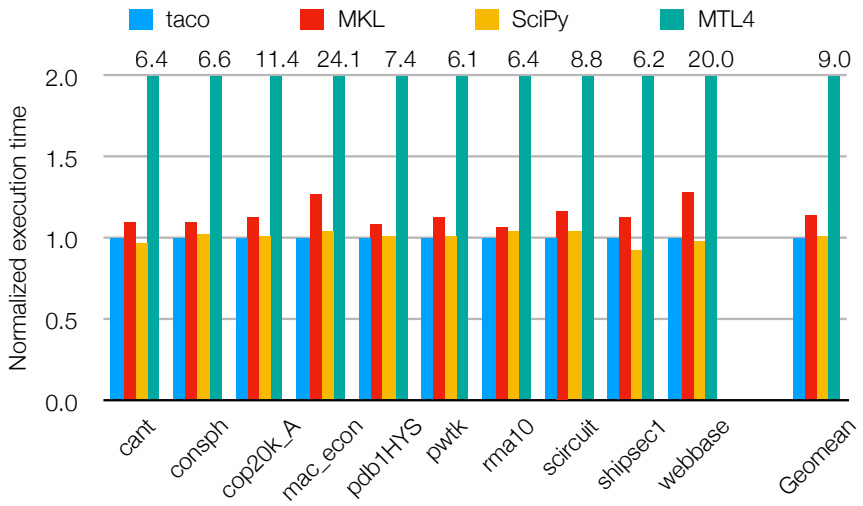


Figure 7-4: Normalized execution times of CSR matrix addition with code that our technique generates (taco) and with other libraries that support the operation. Results are normalized to that of our technique for each matrix.

DIA and ELL SpMV

Code that our technique generates for DIA SpMV iterates over each non-empty matrix diagonal and, for each diagonal, accumulates the element-wise product of the diagonal and the input vector into the result. SciPy implements the same algorithm that our technique generates and thus has the same performance. MKL, by contrast, tiles the computation to maximize cache utilization and thus outperform both our technique and SciPy, particularly for large matrices with many non-empty diagonals. Senanayake et al. [96] show how TACO can be extended with a scheduling language that supports optimizations such as iteration space tiling, though their implementation only supported the `dense` and `compressed` level formats. Future work includes extending the scheduling language to support arbitrary level formats, which would enable our technique to apply the same tiling optimization that MKL employs for DIA SpMV.

Similarly, our technique generates code for ELL SpMV that iterates over stored nonzeros in the order they are laid out in memory. MTL4, on the other hand, iterates over nonzeros row by row to maximize the cache hits for vector accesses. For matrices with few nonzeros per row, this lets MTL4 marginally outperform code that our technique generates. Again, extending TACO's scheduling language to support arbitrary level formats would enable our technique to apply iteration space tiling in order to generate code that iterates over

nonzeros in the same way as MTL4. That said, when the number of nonzeros in each row is large, the approach that MTL4 takes actually reduces the cache hit rate for matrix accesses, since nonzeros that are close in memory are not accessed consecutively. The performance impact of this can be significant, as the results for the cant matrix show.

7.1.3 Sparse Higher-Dimensional Tensor Computations

We also evaluated code that our technique generates against equivalent hand-implemented code in the Tensor Toolbox (TTB) and TensorFlow (TF) for the following sparse higher-dimensional tensor algebra 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	$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

where all three-dimensional tensors and the outputs of TTV, TTM, and PLUS are stored in the COO format with nonzeros stored in coordinate order, while all other operands are dense. All these operations have real-world applications. TTM and MTTKRP, for example, are building blocks of widely used algorithms for computing Tucker and CP decompositions [67, 103]. These same operations were also evaluated in Kjolstad et al. [54, Section 8.4], though that work measured the performance of TACO-generated code that compute with the more efficient CSF format.

Table 7.2 shows the results of our experiments, with Intel MKL, SciPy, and MTL4 omitted as they do not support sparse higher-dimensional tensors. The Tensor Toolbox and TensorFlow are on opposite sides of the trade-off space for hand-written sparse tensor algebra libraries. The Tensor Toolbox supports all the operations in our benchmark but has poor performance, while TensorFlow supports only one operation but is more efficient than the Tensor Toolbox Our technique, by contrast, emits efficient code for all five operations, demonstrating that generality and performance are not mutually exclusive.

Table 7.2: Execution times (in milliseconds) of sparse higher-dimensional tensor algebra kernels with operands stored in COO. Figures in parentheses show speedups that our technique achieves relative to the other libraries. A missing entry means a library does not support an operation, while OOM means the kernel runs out of memory. NELL-2 and NELL-1 are too large for TensorFlow's protocol buffers, and so we omit results for TensorFlow with those tensors.

Kernel	Facebook			NELL-2		NELL-1	
	TACO	TTB	TF	TACO	TTB	TACO	TTB
TTV	13	55 (4.1×)		337	4797 (14.3×)	2253	11239 (5.0×)
TTM	444	18063 (40.7×)		5350	48806 (9.1×)	56478	OOM
PLUS	37	539 (14.6×)	60 (1.6×)	3085	73380 (23.8×)	6289	123387 (19.6×)
MTTKRP	44	364 (8.4×)		3819	43102 (11.3×)	21042	110502 (5.3×)
INNERPROD	12	670 (57.1×)		416	82748 (199.0×)	985	148592 (150.9×)

As with sparse matrix addition, code that our technique generates for adding three-dimensional COO tensors has better performance than TensorFlow's generic sparse tensor addition kernel. Furthermore, code that our technique generates significantly outperform equivalent code that are implemented in the Tensor Toolbox, often by more than an order of magnitude. This is because the Tensor Toolbox relies on MATLAB primitives that cannot always directly operate on arrays storing the input tensors or exploit properties of how the input tensors are stored in order to optimize the computation. To add two sparse tensors, for instance, the Tensor Toolbox computes the set of output nonzeros by calling a MATLAB built-in function that computes the union of the input tensors' nonzeros. MATLAB's implementation of set union, however, cannot exploit the fact that the input tensors are already individually sorted and must sort the concatenation of their nonzeros. By contrast, our technique generates code that directly iterates over and merges the two input tensors without first re-sorting them, reducing the asymptotic complexity of the computation. Additionally, code that our technique generates directly assembles sparse output tensors, whereas, for computations such as TTM, the Tensor Toolbox stores the results in intermediate dense structures.

7.1.4 Benefits of Supporting Disparate Formats

Table 7.3 shows, for a wide variety of array-based sparse tensor formats, which formats are supported by our technique and the other existing sparse linear and tensor algebra libraries

Table 7.3: Support for various (array-based) sparse tensor formats by our technique (TACO) as well as by other existing sparse linear and tensor algebra libraries. (✓) identifies tensor formats that our technique can support by defining additional level formats on top of those described in Section 3.1.

Tensor Type	Format	TACO	MKL	SciPy	MTL4	TTB	TF
Vector	Sparse array	✓	✓	✓	✓	✓	✓
	Hash map	✓		✓			
	Byte map	✓					
Matrix	COO	✓	✓	✓	✓	✓	✓
	CSR	✓	✓	✓	✓	✓	
	DCSR	✓					
	CSC	✓	✓	✓	✓	✓	
	Ragged array	✓					
	Byte map	✓					
	ELL	✓			✓		
	DIA	✓	✓	✓			
	BCSR	✓	✓	✓	✓		
	DOK	(✓)		✓			
	LIL	(✓)		✓			
	SKY	(✓)	✓				
	Banded format	(✓)			✓		
Three-dimensional tensor	COO	✓				✓	✓
	CSR	✓					
	CSF	✓					
	Mode-generic format	✓					

we evaluate. In addition to the formats described in Section 2.1, we also considered all other formats for storing unfactorized, non-symmetric sparse tensors that are supported by at least one existing library. This includes DOK [111], which stores nonzeros using a hash map that is indexed by both the row and column coordinates of nonzeros, LIL [112], which stores each row of a matrix using a separate (array-based) list, and the skyline (SKY) format [90], which is designed for storing variably-banded triangular matrices. Meanwhile, the (sparse) banded matrix format is similar to DIA but instead stores elements in each row contiguously. As Table 7.3 shows, each of the other existing libraries only supports a subset of the tensor formats we consider. By contrast, our technique supports all of the tensor formats we consider, either as shown in Figure 3-3 or by defining additional level formats on top of those described in Section 3.1.

Our technique's support for a wider range of formats can enable it to achieve better performance in practical use, depending on characteristics of the application. The COO

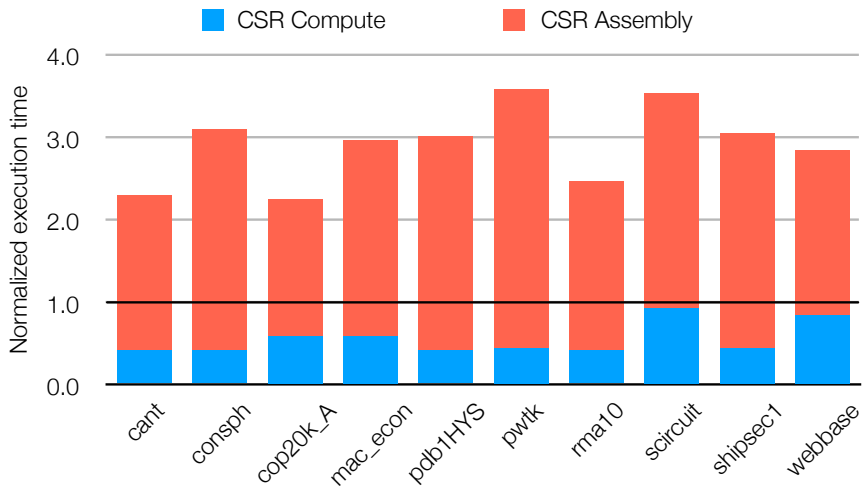


Figure 7-5: Normalized execution times of CSR SpMV relative to COO SpMV, taking into account the cost of converting the input matrix (assumed to be in COO originally) to CSR. These results show that computing with CSR is faster than with COO (black line) only if the cost of converting the input matrix to CSR can be amortized over multiple iterations.

format, for instance, is the intuitive way to represent sparse tensors and is used by many file formats to encode sparse tensors. Thus, it is a natural format for importing and exporting sparse tensors into and out of an application. As the blue bars in Figure 7-5 show, computing matrix-vector products directly on COO matrices can take up to twice as much time as with CSR matrices due to higher memory traffic. If a matrix is imported into the application in the COO format though, then it must be converted to a CSR matrix before the more efficient CSR SpMV kernel can be used. This preprocessing step incurs significant overhead that, as the red bars in Figure 7-5 show, exceeds the cost of computing on the original COO matrix. For non-iterative applications that cannot amortize this conversion overhead, our technique can optimize end-to-end performance by enabling SpMV to be computed directly on the COO matrix, thereby eliminating the overhead.

Supporting a wider range of formats also enables our technique to better exploit characteristics of the input data for performance. To show this, we compared the performance of SpMV computed on CSR and DIA matrices using code that our technique generates. For matrices like Lin and synth1 whose nonzeros are all contained in a few densely-filled diagonals, storing them in DIA exposes opportunities for vectorization. As Figure 7-6 shows, our technique can exploit this to improve the performance of SpMV by up to 22%

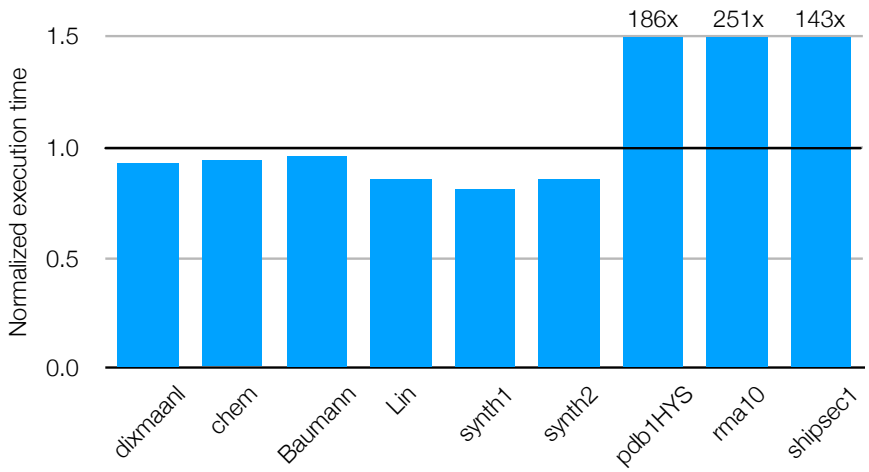


Figure 7-6: Normalized execution times of DIA SpMV code that our technique generates relative to CSR SpMV. Storing the input matrix in the DIA format can accelerate SpMV if all the nonzeros in the matrix are confined to a few densely-filled diagonals, but can drastically degrade performance if that is not the case.

relative to computing the same operation on CSR matrices. On the other hand, DIA SpMV performs significantly worse for matrices like rma10 whose nonzeros are spread amongst many sparsely-filled diagonals, since the kernel has to redundantly compute with all the zeros in the non-empty diagonals.

We further compared the performance of CSR SpMV with input vectors stored as dense arrays, sparse arrays, and hash maps, for operands of varying density. Figure 7-7 shows the results of our experiments. When the input vector contains mostly nonzeros, dense arrays are suitable for SpMV as they provide efficient random access without needing to explicitly store coordinates of nonzeros. Conversely, when the input vector contains mostly zeros and the matrix is much denser, sparse arrays offer better performance for SpMV as it can be computed without accessing all matrix nonzeros. However, when the input vector is large and sparse but still denser than the matrix, computing SpMV with hash map vectors reduces the number of accesses that go out of cache. At the same time, the hash map format's random access capability makes it possible to compute SpMV without accessing the full input vector. By supporting all these different formats for storing vectors, our technique can efficiently compute SpMV regardless of how sparse the operands are.

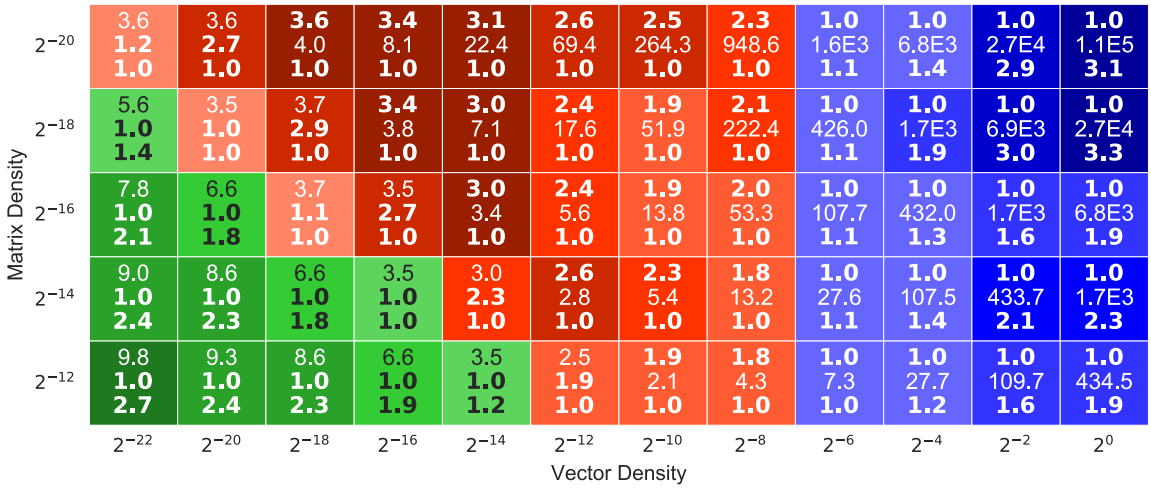


Figure 7-7: Normalized execution times of CSR SpMV with inputs of varying density and input vectors stored in different formats. Results are normalized to that of the most performant format for each configuration. Each cell shows results for dense arrays (top), sparse vectors (middle), and hash maps (bottom). Cells are highlighted based on which vector format is most performant (blue for dense arrays, green for sparse vectors, red for hash maps), with darker shades indicating more significant performance differences.

7.2 Assembling Results in Array-Based Formats

All of the kernels we evaluated in the previous section either have dense outputs or have sparse outputs that can be assembled in order. In this section, I demonstrate our technique's ability to generate efficient code for performing operations with outputs that are sparse and that must also be assembled out of order in array-based formats. In particular, we first evaluated the performance of code that our technique generates for converting sparse tensors between arbitrary combinations of disparate source and target formats. We then evaluated our technique's ability to generate efficient parallel sparse tensor algebra kernels with sparse outputs, which typically must store result nonzeros out of order.

7.2.1 Experiment Setup

To evaluate sparse tensor conversion routines that our technique generates, we compared against equivalent hand-implemented versions in SPARSKIT [93], a widely used [81, 46] Fortran sparse linear algebra library, and Intel MKL. Additionally, to evaluate parallel code that our technique generates for computing sparse tensor algebra operations with sparse

outputs, we compared against SuiteSparse:GraphBLAS [29], a highly-optimized library that can be used to implement graph algorithms in the language of linear algebra.

All experiments were run on a two-socket, 12-core/24-thread 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. For our evaluation of sparse tensor format conversion, we compiled code that our technique generates using GCC 7.5.0 with `-O3 -march=native -mtune=native` optimizations enabled and built SPARSKIT from source using GFortran 7.5.0 with the same optimizations. We ran each experiment 50 times under cold cache conditions report median serial execution times. For our evaluation of parallel sparse algebra tensor computations with sparse outputs, we compiled code that our technique generates using GCC 7.5.0 with `-O3` optimizations enabled. We ran each experiment 100 times under warm cache conditions and report average execution times when run using 12 threads, with execution restricted to a single socket using `numactl`.

We ran our experiments with real-world matrices of varying sizes and sparsity structures as inputs. Table 7.4 shows the matrices we used to evaluate sparse tensor format conversion, while Table 7.8 shows the matrices we used to evaluate parallel sparse tensor algebra computations with sparse outputs. These matrices come from applications in disparate domains and were obtained from the SuiteSparse Matrix Collection.

7.2.2 Sparse Tensor Format Conversion

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

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

Table 7.4: Statistics about matrices used to evaluate code that our technique generates for assembling sparse outputs out of order. Non-symmetric matrices are highlighted in gray.

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

where inputs and outputs in COO, CSR, or CSC are not assumed to be sorted (though nonzeros are still necessarily grouped by row/column in CSR/CSC). For all these combinations of formats, our technique generates code that performs the conversion by assembling the output matrix out of order in the desired target format.

For each combination of formats, we also measured the performance of conversion between those formats using SPARSKIT and MKL. Both libraries implement routines that directly convert matrices from COO to CSR, CSR to CSC, and CSR to DIA. Additionally, SPARSKIT supports directly converting matrices from CSR to ELL. However, neither SPARSKIT nor MKL implements routines that directly convert matrices between the remaining combinations of formats. Thus, to perform those conversions using either library, we had to first convert the input matrix from its source format to a CSR temporary and then convert the CSR temporary to the intended target format. (If the input matrix is symmetric though, then conversions from CSC to DIA/ELL are cast as conversions from CSR to DIA/ELL, since CSR and CSC are equivalent in that case.)

Table 7.5: Performance of sparse tensor format conversion routines that our technique generates (TACO) and that are directly implemented in SPARSKIT or MKL. For each combination of source and target formats, we show absolute execution times of code generated by our technique and speedups that our technique achieves over the other libraries. For CSR to CSC conversion, we only show results for nonsymmetric matrices (rows highlighted in gray) since CSR and CSC are equivalent for symmetric matrices. For conversions to DIA/ELL, we also omit results for matrices that would have to be stored with more than 75% of values being zeros, since having to compute with so many zeros would likely eliminate any performance benefit of DIA/ELL.

Matrix	coo_csr			csr_csc			csr_dia			csr_ell	
	TACO (ms)	SPARSKIT / TACO	MKL / TACO	TACO (ms)	SPARSKIT / TACO	MKL / TACO	TACO (ms)	SPARSKIT / TACO	MKL / TACO	TACO (ms)	SPARSKIT / TACO
1	57.5	1.02	1.11							79.1	1.68
2	0.96	0.97	1.56				0.91	1.85	1.56	0.92	0.95
3	0.93	1.00	1.60				0.91	1.84	1.54	0.82	1.04
4	1.06	1.04	1.44	1.10	0.98	2.14	0.93	1.83	1.54	0.91	0.91
5	34.0	1.01	0.96	29.1	1.17	1.16				49.2	1.84
6	1.61	1.02	1.42				1.54	1.88	1.57	1.35	0.97
7	27.4	1.00	1.35				44.5	3.61	3.63	59.8	1.78
8	1.69	1.00	1.50	1.64	1.07	2.36	1.86	1.85	1.54	1.77	0.94
9	64.8	1.01	1.21							88.9	1.45
10	5.61	1.01	1.51				4.83	2.21	2.26	5.17	1.02
11	4.70	0.99	1.49	4.71	1.03	1.89	3.56	1.95	1.70	3.33	1.01
12	63.6	0.89	0.96							34.8	3.60
13	81.6	1.02	1.28							112	1.93
14	12.3	1.00	1.33	12.1	0.99	1.78	9.91	2.43	2.42	8.17	1.03
15	15.8	1.00	1.10	16.4	0.95	1.09					
16	11.1	0.99	1.29	11.6	1.00	1.38					
17	121	1.00	1.29							123	4.09
18	9.88	0.98	1.36				8.14	1.92	1.70	10.1	0.98
19	42.3	0.99	1.41				35.8	1.64	1.44	37.5	1.08
20	57.9	1.01	0.99	59.3	1.00	1.14					
21	113	0.96	1.21	113	0.97	1.04	62.2	1.72	1.58	74.1	1.17
Geomean		1.00	1.29		1.02	1.48		2.01	1.80		1.36

Tables 7.5 and 7.6 show the results of our experiments. As these results demonstrate, code that our technique generates outperform or are comparable to SPARSKIT and Intel MKL on average for all combinations of source and target formats that we evaluate. On the whole, code that our technique generates for converting matrices from COO to CSR (`coo_csr`) and from CSR to CSC (`csr_csc`) exhibit similar performance as hand-implemented routines in SPARSKIT and exhibit somewhat better performance than Intel MKL. This is unsurprising since our technique generates code that implement the same algorithms (i.e., variations of Gustavson's HALFPERM algorithm [40]) as SPARSKIT. Our technique

Table 7.6: Performance of sparse tensor format conversion routines that our technique generates (TACO) but that are not directly implemented in SPARSKIT or MKL. To perform these conversions using the other libraries, we first convert each matrix to a CSR temporary and then convert the temporary to the intended target format. For each combination of source and target formats, we show absolute execution times of code generated by our technique and speedups that our technique achieves over the other libraries. For symmetric matrices, we cast CSC to DIA/ELL conversion as CSR to DIA/ELL conversion and report the same results shown in Table 7.5. We also omit results for matrices that would have to be stored with more than 75% of values being zeros, since having to compute with so many zeros would likely eliminate any performance benefit of DIA/ELL.

Matrix	coo_dia			csc_dia			csc_ell	
	TACO (ms)	SPARSKIT / TACO	MKL / TACO	TACO (ms)	SPARSKIT / TACO	MKL / TACO	TACO (ms)	SPARSKIT / TACO
1							79.1	1.68
2	0.86	3.07	3.38	0.91	1.85	1.56	0.92	0.95
3	0.86	3.02	3.36	0.91	1.84	1.54	0.82	1.04
4	0.88	4.60	4.94	0.96	4.24	3.85	1.24	3.11
5							62.8	2.09
6	1.50	5.03	3.11	1.54	1.88	1.57	1.35	0.97
7	45.3	4.16	4.39	44.5	3.61	3.63	59.8	1.78
8	1.76	4.98	3.05	1.89	4.68	3.44	2.36	3.00
9							88.9	1.45
10	5.14	5.10	5.78	4.83	2.21	2.26	5.17	1.02
11	3.48	5.23	5.48	3.60	5.07	4.10	4.74	3.16
12							34.8	3.60
13							112	1.93
14	10.9	3.34	3.70	10.4	3.47	4.37	20.1	1.88
14							123	4.09
18	8.40	4.87	5.12	8.14	1.92	1.70	10.1	0.98
19	36.8	2.74	3.00	35.8	1.64	1.44	37.5	1.08
21	64.9	3.26	3.49	62.9	3.40	3.39	88.7	2.20
Geomean		4.01	3.96		2.75	2.51		1.78

also generates code for performing CSR to DIA conversion (`csr_dia`) that is 2.01× faster than SPARSKIT and 1.80× faster than Intel MKL on average. SPARSKIT's implementation of `csr_dia` supports extracting a bounded number of nonzero diagonals from the input matrix and storing them in the output. However, it implements this capability using an inefficient algorithm to identify the densest diagonals, resulting in the slowdown. In addition, `csr_ell` code that our technique generates is 1.36× faster than SPARSKIT on average. This is because our technique generates code that invokes `calloc` to both allocate and zero-initialize the output arrays. SPARSKIT, by contrast, takes user-allocated output arrays as arguments and separately initializes those arrays.

Furthermore, for conversions from COO or CSC to DIA or ELL, code that our technique generates achieve even greater speedups—between 1.78 and 4.01×—over SPARSKIT and Intel MKL . This is because when the input matrix is nonsymmetric or stored in COO, both libraries incur additional memory accesses in order to first construct CSR temporaries and then iterate over those temporaries to convert them to the intended target formats. This show the benefit of a compiler technique such as ours that can generate code to directly construct the outputs of sparse tensor conversion in their desired target formats.

7.2.3 Parallel Computation with Sparse Output

To evaluate our technique's ability to generate parallel sparse tensor algebra code that produce sparse outputs, we measured the performance of code that our technique generates for computing two variants of sparse matrix multiplication, namely

- **sparse Boolean matrix multiplication** ($\forall_i \forall_j \forall_k \; A_{ik} \vee= B_{ij} \wedge C_{jk}$) and
- **sparse min-plus matrix multiplication** ($\forall_i \forall_j \forall_k \; D_{ik} \min= (F_{ij} + G_{jk})$)

where A, B, and C store Boolean elements, D, F, and G store floating-point elements with infinities (instead of zeros) being compressed out, and all matrices are stored in CSR. These kernels can be used to implement many graph algorithms, such as those for performing (multiple-source) breadth-first search and for solving the all-pairs shortest paths problem, in the language of linear algebra [50].

To be able to compute sparse matrix multiplication in parallel and produce a sparse output, our technique generates code that performs the computation in two passes. In the first pass, the generated code performs the computation symbolically in order to compute the number of defined elements[1] in each row of the result and preallocate the output arrays accordingly. Then, in the second pass, the generated code performs the computation again numerically and appends each defined element in the result to the end of its corresponding segment in the output arrays. This approach enables multiple

[1] *Defined elements* generalize the notion of nonzeros in cases where non-zero fill values are compressed out, as described in Section 3.4. In the case of Boolean matrix multiplication, false elements are the compressed-out fill values while true elements are considered defined. Meanwhile, in the case of min-plus matrix multiplication, the infinite elements are fill values while finite elements are considered defined.

Table 7.7: Performance of parallel sparse matrix multiplication code that our technique generates (TACO) and that are implemented in SuiteSparse:GraphBLAS. For each kernel, we show absolute execution times of code generated by our technique and speedups that our technique achieves over SuiteSparse:GraphBLAS. For SuiteSparse:GraphBLAS, we report results for whichever algorithm the library chooses (A) as well as for their implementation of Gustavson's algorithm (G), which uses dense arrays to store partial results. We use each matrix as both inputs to each kernel.

Matrix	Boolean Matrix Multiplication			Min-Plus Matrix Multiplication		
	TACO (ms)	$\frac{\text{SuiteSparse (A)}}{\text{TACO}}$	$\frac{\text{SuiteSparse (G)}}{\text{TACO}}$	TACO (ms)	$\frac{\text{SuiteSparse (A)}}{\text{TACO}}$	$\frac{\text{SuiteSparse (G)}}{\text{TACO}}$
1	35.7	0.896	3.50	67.5	0.599	2.46
2	815	0.804	1.59	1414	0.577	1.28
3	55.1	1.40	1.37	104	1.14	1.23
4	2203	1.12	1.12	4510	0.946	0.946
6	1897	1.05	1.09	2607	0.955	1.02
7	120754	0.997	0.997	276389	0.748	0.748
8	6336	1.23	1.02	9595	1.13	1.01
9	76.6	0.836	0.998	121	0.687	0.825
10	1105	0.852	1.29	1678	0.661	1.05
11	1127	0.689	0.987	1733	0.558	0.819
12	109	1.28	1.06	180	1.31	1.10
13	17184	1.32	1.11	36121	1.17	1.07
Geomean		1.02	1.25		0.836	1.07

threads to simultaneously compute different rows of the output matrix without interfering with each other. The generated code has similar high-level structure as code shown in Figure 6-4c for converting a matrix to CSR, except that instead of simply copying over elements from the input matrix, the elements that are inserted into the output are computed on-the-fly. SuiteSparse:GraphBLAS, which we compared our generated code against, also parallelizes sparse matrix multiplication in a similar way.

Table 7.7 show the results of our experiments. Code that our technique generates is 1.02× faster than SuiteSparse:GraphBLAS on average for sparse Boolean matrix multiplication and has performance 0.836× that of SuiteSparse:GraphBLAS on average for sparse min-plus matrix multiplication. The similarities in performance can be explained by the fact that our generated code and SuiteSparse:GraphBLAS both use a linear combination of rows algorithm to compute the kernels. However, when the output matrix contains relatively few defined elements per row, SuiteSparse:GraphBLAS is able to use hash tables to store partial results. By contrast, our technique currently can only generate code that

stores partial results using dense arrays, which reduces cache efficiency. Table 7.7 also shows, though, that code our technique generates has similar or better performance on average than SuiteSparse when the latter also uses dense arrays to store partial results (which is preferable when the output is relatively dense).

7.3 Computing with Pointer-Based Formats

To conclude this chapter, I demonstrate our technique's ability to also generate efficient code for computing on sparse tensors that are stored in pointer-based formats.

7.3.1 Experiment Setup

We evaluated code that our technique generates against four state-of-the-art libraries and frameworks: Aspen [32], Terrace [79], STINGER [34], and PAM [109]. Aspen and Terrace are graph processing frameworks that let users compute on dynamic graphs by invoking a fixed set of hand-optimized primitives for mapping over and applying user-defined functions on edges and vertices. Internally, Aspen stores graphs as adjacency matrices using C-trees, while Terrace uses a combination of fixed-size arrays, packed-memory arrays, and B-trees. STINGER is another graph processing framework that supports computations on dynamic graphs; STINGER stores graphs using block linked lists and provides a set of macros that programmers can use to iterate over (and compute on) edges in a graph. PAM, by contrast, is a lower-level, parallel C++ library that implements a fixed set of primitives for operating on ordered key-value maps stored as self-balancing BSTs. While PAM does not directly implement any tensor algebra kernel, the primitives that PAM exposes can be utilized to perform computations on sparse tensors that are stored using BSTs.

All experiments were run on a two-socket, 12-core/24-thread 2.5 GHz Intel Xeon E5-2680 v3 machine with 30 MB of L3 cache per socket and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. We compiled all code using GCC 7.5.0 with `-O3 -march=native -mtune=native -ffast-math` optimizations enabled. To ensure apples-to-apples comparisons of the actual algorithms that are implemented by the (generated and hand-optimized) kernels, we modified code generated by our technique so that they operate

Table 7.8: Statistics about matrices used to evaluate code that our technique generates for computing on operands stored in pointer-based formats.

	Matrix	Dimensions	NNZ
1	belgium_osm	1.44M × 1.44M	3.10M
2	cit-Patents	3.77M × 3.77M	16.5M
3	coAuthorsCiteseer	227K × 227K	1.63M
4	coPapersDBLP	540K × 540K	30.5M
5	com-Orkut	3.07M × 3.07M	234M
6	delaunay_n24	16.8M × 16.8M	101M
7	indochina-2004	7.41M × 7.41M	194M
8	rgg_n_2_24_s0	16.8M × 16.8M	265M
9	roadNet-CA	1.97M × 1.97M	5.53M
10	road_central	14.1M × 14.1M	33.9M
11	road_usa	23.9M × 23.9M	57.7M
12	ship_003	122K × 122K	3.78M
13	soc-LiveJournal1	4.85M × 4.85M	69.0M
14	webbase-1M	1.00M × 1.00M	3.11M

on the exact same data structures in memory as the libraries we compare against. This only required minor changes to how the fields of input data structures are accessed and did not entail any algorithmic change. Additionally, we used the same parallel programming APIs as the libraries we compare against (i.e., OpenMP for STINGER and Cilk for the others), and all memory allocations were done using `jemalloc`. We ran each experiment 100 times under cold cache conditions and report median execution times. Each experiment was run using 24 threads, with execution restricted to a single socket using `numactl`.

We ran our experiments with real-world sparse matrices of varying sizes from the SuiteSparse Matrix Collection. These matrices, which Table 7.8 describes in more detail, represent graphs and other data that arise in disparate application domains.

7.3.2 Support for Disparate Formats

Our technique generates efficient code for computing on sparse tensors that are stored in a wide range of disparate pointer-based formats. To demonstrate this, we evaluated the performance of code that our technique generates for performing

- the **PageRank** kernel ($\forall_i \forall_j \; y_i \mathrel{+}= A_{ij} x_j d_j^{-1}$) and
- **sparse matrix-dense matrix multiplication** ($\forall_i \forall_j \forall_k \; C_{ik} \vee= A_{ij} \wedge B_{jk}$, or **SpMM**)

where A is a sparse Boolean matrix (representing a graph's adjacency matrix), B and C are dense Boolean matrices, x and y are dense floating-point vectors, and d is a dense integer vector. In particular, the first kernel corresponds to the main kernel in each iteration of the PageRank algorithm [78], while the second kernel can be used to implement algorithms such as multi-source breadth-first search [3]. For each kernel, we measured the performance of code that our technique generates for A stored in various pointer-based tensor formats (specified in terms of level formats that are defined in Figures 5-2 and 5-3), including

- using only **BSTs** in the (`bst`, `bst`) format,
- using **C-trees** in the (`bst`, `ctree`) format,
- using **block linked lists** (**BLLs**) in the (`dense`, `blist`) format, and
- using a **hybrid of fixed-size arrays and B-trees** in the (`dense`, `hybrid`) format;

these correspond to formats that are supported by PAM, Aspen,[2] STINGER, and Terrace[3] respectively. We then compare the generated code against the aforementioned frameworks and library. In particular, Aspen and Terrace both implement an `edgeMap` primitive that can compute the PageRank and SpMM kernels by iterating over edges in the input graph (i.e., defined elements in A) and performing some (user-defined) computation on each edge/nonzero. Similarly, STINGER provides macros that can be used to efficiently iterate over incident edges of each vertex (i.e., defined elements in each row of A) in order to perform the same computations. Additionally, PAM can compute the same kernels in similar ways by mapping over the rows of A (using the `map_void` primitive) and performing either a map-reduce operation (for PageRank, using the `semi_map_reduce` primitive) or another map operation (for SpMM) over the defined elements in each row.

Tables 7.9 and 7.10 show the results of our experiments. Our technique is the only one that supports all of the formats we consider; the other frameworks and library we evaluate each only supports a single format. Nevertheless, as Tables 7.9 and 7.10 demonstrate, our technique is able to achieve comparable, if not better, performance than all the hand-

[2]While Aspen also supports C-trees that use difference encoding to compress the coordinates stored in each block, we only evaluate our technique and Aspen on C-trees that do not use difference encoding, since difference encoding is not supported by our technique as we have described it.

[3]Terrace supports packed-memory arrays (PMAs) in addition to fixed-size arrays and B-trees, though we omit PMAs from our evaluation since they are not supported by our technique.

Table 7.9: Performance of code implemented using existing libraries and generated by our technique (TACO) for computing the PageRank kernel on sparse matrices that are stored in disparate pointer-based formats. For each format, we show execution times of code implemented using an existing library that supports the format, execution times of code generated by our technique, and speedups achieved by the generated code. Each test matrix is identified by its label as shown in Table 7.1.

Matrix	BSTs			C-trees			BLLs			Array/B-tree Hybrid		
	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$	Aspen (ms)	TACO (ms)	$\frac{\text{Aspen}}{\text{TACO}}$	STINGER (ms)	TACO (ms)	$\frac{\text{STINGER}}{\text{TACO}}$	Terrace (ms)	TACO (ms)	$\frac{\text{Terrace}}{\text{TACO}}$
1	9.876	5.508	1.793	6.970	6.310	1.105	15.05	14.18	1.061	3.080	2.783	1.107
2	61.51	43.53	1.413	49.23	36.96	1.332	63.77	56.96	1.120	40.46	32.28	1.253
3	3.142	2.506	1.254	1.712	2.018	0.848	3.314	2.951	1.123	1.766	1.100	1.605
4	31.80	27.70	1.148	14.99	12.30	1.219	34.87	28.89	1.207	26.96	14.36	1.877
5	504.0	449.7	1.121	378.7	276.4	1.370	494.3	403.3	1.226	471.5	288.0	1.637
6	143.8	111.6	1.288	85.48	76.36	1.119	218.4	199.0	1.098	46.12	42.40	1.088
7	188.7	181.6	1.040	90.45	70.85	1.277	195.6	171.5	1.141	529.9	164.6	3.220
8	378.9	265.8	1.426	163.0	147.1	1.108	435.5	355.5	1.225	306.7	148.1	2.071
9	12.54	7.817	1.604	8.852	8.089	1.094	24.19	20.35	1.189	3.920	3.582	1.094
10	172.9	98.67	1.752	117.5	96.22	1.221	192.5	184.7	1.042	71.48	62.27	1.148
11	166.1	95.12	1.746	118.1	106.5	1.109	269.1	258.7	1.040	58.86	52.32	1.125
12	7.044	6.734	1.046	3.613	3.046	1.186	6.076	5.751	1.056	6.868	3.638	1.888
13	156.2	124.1	1.259	103.2	80.12	1.288	176.7	149.7	1.180	157.3	79.48	1.979
14	6.715	4.811	1.396	5.220	4.005	1.303	9.224	8.707	1.059	4.879	2.738	1.782
Geomean			1.355			1.177			1.124			1.551

optimized frameworks and library. In particular, code that our technique generates for computing on C-trees and block linked lists have similar performance as Aspen and STINGER, with the generated code being 1.144–1.177× faster than Aspen and 1.002–1.124× faster than STINGER on average. Additionally though, code that our technique generates for computing on B-trees outperform Terrace by 1.214–1.551× on average, while code that our technique generates for computing on BSTs outperforms PAM by 1.202–1.355× on average. It is not surprising that our technique achieves similar performance as Aspen and STINGER, since code our technique generates essentially implement the same high-level algorithms as Aspen and STINGER. Meanwhile, Terrace is slower than our technique since its implementation of `edgeMap` traverses B-trees using a sequential iterator that has more complicated control flow, which increases the cost of accessing each nonzero. By contrast, for both PageRank and SpMM, our technique can generate code that instead recursively traverses the B-trees, which reduces the overhead of accessing nonzeros.

Table 7.10: Performance of code implemented using existing libraries and generated by our technique (TACO) for computing SpMM on matrices that are stored in disparate pointer-based formats. For each format, we show execution times of code implemented using an existing library that supports the format, execution times of code generated by our technique, and speedups achieved by the generated code. Each test matrix is identified by its label as shown in Table 7.1.

Matrix	BSTs			C-trees			BLLs			Array/B-tree Hybrid		
	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$	Aspen (ms)	TACO (ms)	$\frac{\text{Aspen}}{\text{TACO}}$	STINGER (ms)	TACO (ms)	$\frac{\text{STINGER}}{\text{TACO}}$	Terrace (ms)	TACO (ms)	$\frac{\text{Terrace}}{\text{TACO}}$
1	23.40	16.95	1.380	20.19	17.06	1.184	27.19	29.33	0.927	16.23	14.60	1.112
2	163.4	153.3	1.066	138.3	126.2	1.095	147.9	150.2	0.985	146.8	124.6	1.178
3	11.21	9.839	1.140	8.825	8.374	1.054	11.85	11.47	1.033	9.378	8.408	1.115
4	122.5	99.96	1.225	87.24	78.24	1.115	108.4	102.0	1.062	109.8	90.58	1.213
5	1056	910.1	1.161	769.0	676.8	1.136	947.3	928.2	1.021	900.0	759.1	1.186
6	846.6	704.8	1.201	721.3	660.1	1.093	759.3	753.6	1.008	703.0	652.7	1.077
7	771.8	691.4	1.116	585.1	559.9	1.045	674.6	627.3	1.075	2295	963.1	2.383
8	1823	1634	1.115	1425	1346	1.058	1601	1558	1.028	1620	1388	1.167
9	43.82	33.83	1.295	37.20	32.73	1.136	48.09	51.74	0.930	33.53	30.65	1.094
10	344.4	282.0	1.222	302.4	269.9	1.121	337.8	368.4	0.917	252.1	227.9	1.106
11	457.1	338.3	1.351	391.8	329.5	1.189	477.9	516.1	0.926	333.5	300.4	1.110
12	30.13	25.35	1.189	21.68	20.40	1.063	28.93	26.61	1.087	28.24	22.62	1.248
13	413.6	366.0	1.130	320.3	288.9	1.109	374.0	370.8	1.009	401.2	307.1	1.307
14	17.55	13.70	1.281	20.21	11.46	1.764	17.92	17.29	1.037	18.70	17.07	1.096
Geomean			1.202			1.144			1.002			1.214

7.3.3 Support for Disparate Computations

Not only does our technique support many disparate formats though, our technique generates efficient code to compute a wide range of different operations (in addition to those evaluated in Section 7.3.2) on tensors that are stored in those formats. To demonstrate this, we evaluated the performance of code that our technique generates for performing

- **sparse matrix-vector multiplication** ($\forall_i \forall_j\ y_i \mathrel{+}= A_{ij} x_j$, or **SpMV**)

where A is a sparse matrix stored in the (`bst`, `bst`) format, x is a dense vector, and y is a sparse vector stored as a BST, as well as

- **sparse matrix addition** ($\forall_i \forall_j\ D_{ij} = A_{ij} + C_{ij}$, or **SpAdd**),
- **sparse matrix element-wise multiplication** ($\forall_i \forall_j\ D_{ij} = A_{ij} C_{ij}$, or **SpElMul**), and
- **row-wise inner product** ($\forall_i \forall_j\ z_i \mathrel{+}= A_{ij} B_{ij}$, or **RowInnerProd**)

where A and B are dynamic sparse matrices stored in the (`dense`, `bst`) format, C and D are static sparse matrices stored in CSR, and z is a dense vector.

None of the kernels above are readily supported by Aspen, STINGER, or Terrace, regardless of what format is used to store A and B. (More generally, the three frameworks do not readily support computations that store results in pointer-based formats, that simultaneously work with sparse operands in pointer-based and array-based formats, or that perform non-element wise operations on multiple sparse operands.) By contrast, PAM, which is a lower-level library, implements a number of primitives that can be utilized to compute all of the kernels above. In particular, PAM can be used to compute sparse matrix-vector multiplication in a similar way as the PageRank kernel, except that the map operation (`map`) over A's rows also constructs a new BST to store the nonzeros of the output vector.[4] Meanwhile, sparse matrix addition and element-wise multiplication can be computed row by row by having PAM convert each row of A to a BST, compute the union/intersection of the BST with the corresponding row in C (using `map_union`/`map_intersect`), and map over the result (which is also stored in a BST) to copy each result nonzero to D (using `foreach_index`). Additionally, row-wise inner product can be computed by having PAM first compute the intersection of each row of A with its corresponding row in B and then perform a map-reduce over the result in order to compute the corresponding entry in z. We therefore limited our comparisons to PAM and do not consider the other frameworks.

Table 7.11 shows the results of our experiments. As these results demonstrate, our technique generates code that significantly outperforms PAM for all of the kernels we evaluate. In particular, code that our technique generates for adding a matrix stored using BSTs to a matrix stored in CSR outperforms PAM by 6.975× on average. Meanwhile, code that our technique generates for element-wise multiplying a matrix stored using BSTs to a matrix stored in CSR outperforms PAM by 7.224× on average. Performing these computations using PAM incurs significant overhead since the library only supports computing unions and intersections of BSTs. Thus, to compute each row of the result, many additional memory operations are needed in order to allocate new nodes when

[4]While PAM uses a custom pool allocator to allocate new BST nodes by default, we modify PAM for our experiments so that it simply uses `malloc` to allocate new nodes. We find that, for our benchmarks, this slightly improves PAM's performance and also yields more repeatable performance results.

Table 7.11: Performance of code implemented using PAM and generated by our technique (TACO) for computing disparate sparse linear algebra operations on matrices that are stored using BSTs. For each operation, we show execution times of code implemented using PAM, execution times of code generated by our technique, and speedups achieved by the generated code. Each test matrix is identified by its label as shown in Table 7.1.

Matrix	SpMV			SpAdd			SpElMul			RowInnerProd		
	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$	PAM (ms)	TACO (ms)	$\frac{\text{PAM}}{\text{TACO}}$
1	8.78	7.411	1.185	94.65	11.50	8.233	79.46	8.642	9.195	20.95	5.929	3.533
2	54.29	44.30	1.225	390.6	60.65	6.440	304.2	43.62	6.974	113.1	28.82	3.925
3	3.101	2.734	1.134	39.11	5.302	7.377	34.29	4.278	8.016	12.68	3.048	4.161
4	29.73	27.58	1.078	606.5	79.46	7.633	534.1	70.08	7.622	233.0	51.73	4.503
5	375.4	344.9	1.088	5097	653.1	7.805	3516	443.9	7.920	1547	385.1	4.017
6	156.1	132.0	1.182	2162	322.1	6.712	1761	261.6	6.733	614.1	169.1	3.633
7	193.9	188.4	1.029	4262	827.1	5.153	4018	790.5	5.083	1540	497.1	3.098
8	355.1	271.0	1.310	5057	769.2	6.574	3732	511.6	7.294	1592	428.8	3.713
9	11.86	10.48	1.132	148.9	22.55	6.601	123.2	18.24	6.754	35.02	10.13	3.457
10	169.9	115.4	1.473	964.8	144.0	6.698	804.6	111.2	7.234	220.1	61.95	3.553
11	180.8	132.5	1.365	1629	240.5	6.774	1364	192.9	7.073	371.4	105.3	3.528
12	7.040	6.944	1.014	174.4	19.19	9.092	169.1	18.38	9.201	71.26	13.38	5.327
13	132.0	110.9	1.190	1501	207.5	7.233	1137	154.9	7.345	463.4	115.5	4.012
14	6.256	5.511	1.135	100.6	16.28	6.180	81.95	14.06	5.829	25.07	8.645	2.900
Geomean			1.175			6.975			7.224			3.770

converting a row of C (which is stored in CSR) to a BST and also when actually performing the union/intersection operation. Moreover, PAM incurs additional overhead in order to copy nonzeros that are computed by the union/intersection operation over to the output matrix D, which is also stored in CSR. By contrast, our technique generates code that merges the two input matrices by simultaneously iterating over their nonzeros and that directly stores the result nonzeros into the CSR output without needing a BST temporary. This shows the benefits of a compiler technique like ours that can generate efficient code to directly compute with sparse tensors stored in pointer-based and array-based formats.

Furthermore, code that our technique generates for computing the row-wise inner product of two matrices stored using BSTs outperforms PAM by 3.770× on average. PAM again incurs significant overhead since it does not support directly performing a map-reduce operation over the result of an intersection operation, and so computing an inner product requires performing the intersection and map-reduce operations separately. By contrast, our technique is able to generate more efficient code that effectively fuses the

intersection and map-reduce operations. This demonstrates the benefits of a system that does not rely on hand-optimized code to perform a bounded set of operations.

7.3.4 Analysis of Generated Code

We ran several additional experiments—focusing on PageRank with input matrices stored using BSTs as an example—to evaluate the effectiveness of optimizations that our technique applies to generated code. In particular, we compared code that our technique generates, which maps over nonzeros in each row of the input matrix in coordinate order, against code that instead pre-order traverses the BSTs to perform the computation. As Table 7.12 shows, generating map functions that compute on stored nonzeros in coordinate order can yield speedups of 1.063× on average. Furthermore, we compared code that our technique generates, which computes on input matrices using only recursive map functions, against code that instead iterates over input matrices using iterators like those shown in Figure 1-5c. As Table 7.13 shows, by generating recursive map functions to perform the computation where possible, our technique can yield speedups of 1.058× on average. This emphasizes the need for a compiler to be able to generate code that access a sparse tensor's nonzeros in different ways depending on the computation.

7.3.5 Benefits of Supporting Disparate Formats

Again, our technique's ability to work with many disparate formats—including pointer-based as well as array-based formats—can enable it to more effectively support a wider variety of applications with different characteristics. To demonstrate this, we compared the cost of computing on and modifying sparse tensors that are stored using BSTs in the (`bst`, `bst`) format, using C-trees in the (`bst`, `ctree`) format, and in the CSR format. To measure the cost of computing on sparse tensors stored in the aforementioned formats, we measured the performance of code that our technique generates for computing the PageRank kernel. To measure the cost of modifying sparse tensors stored in the same formats, we adapted the method used by Dhulipala et al. [32] and randomly sample 1% of nonzeros in each test matrix to treat as new nonzeros that need to be inserted. We then

Table 7.12: Performance of PageRank kernels that map over input nonzeros out of order (by performing pre-order traversals) and in order. We show execution times of both kernels and show the speedups achieved by the in-order approach. Each test matrix is identified by its label as shown in Table 7.1.

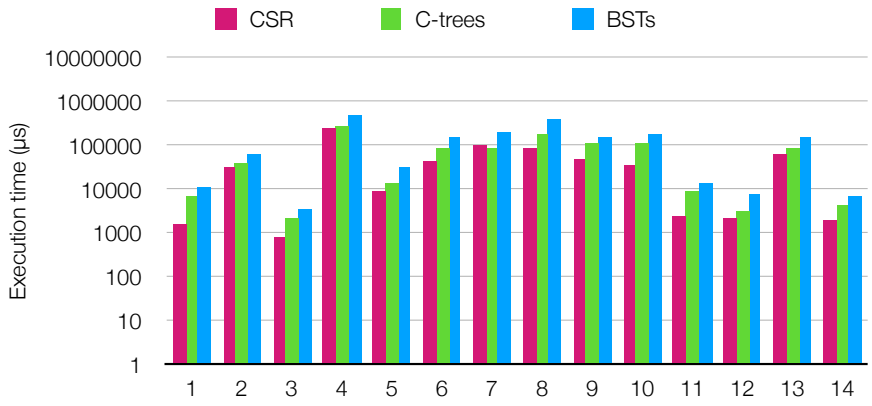
Matrix	Unordered (ms)	Ordered (ms)	$\frac{\text{Unordered}}{\text{Ordered}}$
1	6.233	5.508	1.132
2	44.48	43.53	1.022
3	2.735	2.506	1.092
4	29.03	27.70	1.048
5	441.5	449.7	0.982
6	119.8	111.6	1.073
7	186.2	181.6	1.026
8	332.2	265.8	1.250
9	8.605	7.817	1.101
10	104.3	98.67	1.057
11	106.0	95.12	1.114
12	6.774	6.734	1.006
13	127.6	124.1	1.028
14	4.698	4.811	0.977
Geomean			1.063

Table 7.13: Performance of PageRank kernels that rely on generated iterators and generated recursive map functions. We show execution times of both kernels and show the speedups achieved by the map function approach. Each test matrix is identified by its label as shown in Table 7.1.

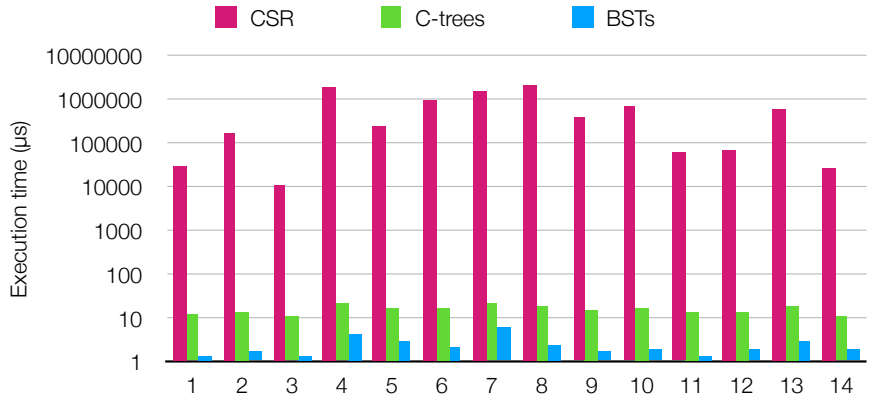
Matrix	w/ Iterator (ms)	w/ Map Func. (ms)	$\frac{\text{w/ Iterator}}{\text{w/ Map Func.}}$
1	6.688	5.508	1.214
2	48.35	43.53	1.111
3	2.727	2.506	1.088
4	27.87	27.70	1.006
5	449.5	449.7	1.000
6	112.7	111.6	1.010
7	181.8	181.6	1.001
8	266.9	265.8	1.004
9	8.491	7.817	1.086
10	113.4	98.67	1.149
11	105.6	95.12	1.110
12	6.850	6.734	1.017
13	129.0	124.1	1.040
14	4.833	4.811	1.005
Geomean			1.058

measured the performance of optimized routines that are implemented in PAM, Aspen, and Eigen (a widely-used sparse linear algebra library that supports CSR [39]) for inserting the sampled nonzeros individually into the tensor.

Figure 7-8 shows the results of our experiments. As these results illustrate, the performance of compute with sparse tensors that are stored in a particular format is often anti-correlated with the performance of modification on sparse tensors that are stored in the same format. In particular, as Figure 7-8a shows, computing the PageRank kernel on CSR matrices is 1.788× faster on average than computing the same kernel on matrices stored using C-trees, which in turn is 1.404× faster on average than computing on matrices stored using BSTs. On the other hand, as Figure 7-8b shows, inserting new nonzeros into matrices stored using BSTs is 6.893× faster on average than inserting new nonzeros into matrices stored using C-trees, which in turn is orders of magnitudes faster than inserting new nonzeros into CSR matrices. These results show how, depending on the relative proportion of data modification and compute, different applications can benefit from using



(a) Compute performance



(b) Insert performance

Figure 7-8: Time required to (a) compute the PageRank kernel on sparse matrices stored in various formats and (b) insert a new nonzero into matrices stored in the same formats. Labels along the horizontal axis identify the input matrices as listed in Table 7.1.

different (pointer-based or array-based) formats to store sparse tensors. This means a system intended to be general-purpose needs to support a wide range of disparate formats.

7.4 Chapter Summary

In this chapter, we saw that sparse tensor algebra code our technique generates has performance comparable to, if not better than, equivalent hand-optimized code implemented in existing libraries and frameworks. In particular, we saw that being able to generate code that directly performs an operation can yield significant performance improvements over approaches that instead require decomposing the operation into multiple kernels. Additionally, we saw that our technique supports a wider range of sparse tensor formats

than existing libraries and frameworks. This makes it possible for our technique to more effectively optimize the performance of sparse tensor computations based on characteristics of the application and the input data.

Chapter 8

Related Works

In this chapter, I review libraries and systems that rely on hand-optimized kernels to perform sparse linear and tensor algebra computations, as well as survey other compiler-based techniques for generating sparse and dense linear and tensor algebra code. Additionally, I survey related works on synthesizing data structure operations from declarative specifications and on compiling declarative query languages, both of which share similarities with various aspects of our technique.

8.1 Sparse Programming Systems and Libraries

State-of-the-art sparse linear and tensor algebra libraries and frameworks [102, 119, 47, 1, 23, 44, 39, 11, 65, 19, 49, 108, 67, 21] typically expose a fixed set of tensor operations that are each implemented as manually-optimized code. This reliance on hand-optimized code, which can be difficult and time-consuming to implement, limits the number of sparse tensor formats that these libraries can support, with most libraries only being able to work with data in one or two formats. In addition, some programming languages such as MATLAB [69] and Julia [14] provide first-class support for sparse linear algebra operations. Again though, these languages typically implement support for sparse linear algebra operations using hand-optimized kernels and suffer from the same limitations as a result.

There also exist sparse tensor algebra libraries and frameworks that, instead of implementing bespoke code for every tensor operation, reduces each operation to a sequence of

lower-level operations and performs each sub-operation by invoking hand-optimized code. As an example, the MATLAB Tensor Toolbox [10] maps sparse tensor operations to MATLAB intrinsics. In particular, to compute sparse tensor-matrix multiplication for instance, the Tensor Toolbox first matricizes the sparse tensor input—storing it as a CSC matrix—and then exploits MATLAB's native support for sparse matrices to perform the computation as a sparse matrix-dense matrix multiplication. Similarly, the Cyclops Tensor Framework (CTF) [104] and libtensor [36] are able to support sparse tensor contractions by transposing tensor operands and performing the tensor contractions as matrix multiplications. Finally, Sparso [91] is a framework that can discover and exploit invariant properties of matrices in a sparse linear algebra program in order to optimize the program as a whole by invoking hand-optimized Intel MKL kernels. However, decomposing a sparse tensor computation into multiple sub-operations typically requires constructing—and then iterating over—large temporaries, which generally incurs significant performance overhead.

Additionally, there exist many libraries and frameworks [34, 61, 98, 32, 79, 68, 97, 121] for implementing graph algorithms. Many real-world computations on graphs can actually be viewed as linear algebra computations that operate on adjacency matrix representations of graphs [50]. As a result, many primitives that are commonly provided by graph processing frameworks resemble sparse linear algebra kernels under the hood. (GraphBLAS libraries such as SuiteSparse:GraphBLAS [29], meanwhile, make this connection explicit by instead simply providing linear algebra operators as primitives in their API.) Again though, all these libraries generally rely on hand-optimized kernels to perform the actual computations.

8.2 Sparse Linear and Tensor Algebra Compilation

Our technique builds on the TACO sparse tensor algebra compiler theory, which was first proposed by Kjolstad et al. [54] and subsequently extended by Kjolstad et al. [53], Senanayake et al. [96], and Henry et al. [42]. Our technique generalizes TACO by enabling it to support sparse tensors that are stored in data structures other than just those used by the CSR format. Donenfeld et al. [33] further generalized our technique to support tensor formats that losslessly compress stored elements.

In addition to TACO though, a number of other researchers have also proposed techniques for generating or optimizing sparse linear and tensor algebra programs. We review these works below.

psi [113] As part of the psi compiler project, Thibault et al. proposed a technique that describes regular geometric partitions in arrays and automatically generates corresponding indexing functions. This technique enables a compiler to generate code that operate on compressed matrices with regular structures, but the technique does not generalize to unstructured sparse matrices.

MT1 [17, 18, 15] Developed by Bik and Wijshoff, the MT1 compiler takes dense linear algebra code as input and transforms it to equivalent sparse code by applying a set of predefined analyses and loop transformations. However, MT1 only supports generating code to compute with a fixed set of standard sparse matrix formats; notably, MT1 does not support higher-dimensional tensor formats or pointer-based formats. Additionally, MT1 does not support computations that require co-iterating over multiple sparse operands.

Bernoulli [59, 107, 58] Developed by Stodghill, Kotlyar, and Pingali, among others, the Bernoulli compiler similarly takes dense code as input and transforms it to equivalent sparse code. Bernoulli achieves this by first lifting dense code to relational algebra, which is then lowered to sparse code by applying a set of predefined rules. To represent different possible sparse matrix formats, Bernoulli uses a *black-box protocol* that, similar to our abstract interface for array-based level formats, hides the details of physical storage behind a fixed interface, which guides the code generation process. In contrast to our technique though, Bernoulli does not support generating efficient code to assemble sparse matrix results from scratch, which is essential for many real-world sparse linear and tensor algebra applications. In addition, Bernoulli does not effectively support most pointer-based sparse tensor formats. While Kotlyar [58] shows how the black-box protocol supports array-based linked lists, they do not consider tree-based data structures like BSTs, for which efficient sequential iterators (which the black-box protocol requires users to define) are not only difficult to implement by hand but also inadequate for generating efficient parallel code.

SIPR [85] Developed by Pugh and Shpeisman, SIPR is another framework that transforms dense linear algebra code to equivalent sparse code. SIPR represents sparse vectors and matrices using hard-coded *element stores* that provide enumerators and accessors, which are analogous to capabilities of level formats in our technique. However, SIPR currently only supports a fixed set of element stores; notably, it does not support structured sparse tensor formats (such as DIA and ELL) or pointer-based sparse tensor formats.

LL [5, 6] Developed by Arnold et al., LL is a verifiable functional language for specifying sparse matrix programs. LL supports various commonly-used sparse matrix formats like CSR and COO, though again it does not pointer-based sparse tensor formats. Moreover, specifications of sparse matrix programs in LL are hard-coded for the formats of the operands and result, so the same sparse matrix operation can require very different specifications for different input/output matrix formats.

CHiLL-I/E [117, 125, 118, 116, 74] Developed by Venkat et al., CHiLL uses polyhedral compilation with uninterpreted functions to generate optimized inspector/executor code for sparse linear algebra computations. CHiLL supports a variety of sparse matrix formats, including structured matrix formats like DIA and BCSR, though again it does not support pointer-based sparse tensor formats. Nandy et al. [74] later extended CHiLL to support automatically generating (without the need for manually-implemented templates) inspectors that can convert input matrices to different sparse matrix formats. However, their approach requires specifications that hard code the specific data structures that are used to store the input matrix in its original format. Thus, their approach cannot support assembling sparse tensor results for computations that take multiple sparse tensor operands as inputs (e.g., sparse matrix multiplication).

MLIR [16] Developed by Bik et al., MLIR's `SparseTensor` dialect is an intermediate representation that provides first-class support for sparse tensor operations. Under the hood, the MLIR compiler uses an implementation of the TACO sparse tensor algebra compiler theory to compile programs that are expressed in the `SparseTensor` dialect to

lower-level code. However, the `SparseTensor` dialect currently only supports sparse tensor formats that can be expressed as compositions of the `dense` and `compressed` level formats.

COMET [114] Developed by Tian et al., COMET is another sparse tensor algebra compiler that is built on top of MLIR. The COMET compiler generates optimized sparse tensor algebra code by progressively lowering the input program to lower-level MLIR dialects. However, COMET only supports sparse tensor formats that can be expressed as compositions of the `dense`, `compressed`, and `singleton` level formats.

8.3 Dense Linear and Tensor Algebra Compilation

Much work has also been done on compilers [105, 76, 120] and loop transformation techniques [123, 122, 72] for dense linear algebra. Array programming languages like Fortran, APL, MATLAB, and Julia provide the ability to operate on dense arrays as a whole using scalar operators. Meanwhile, the Tensor Contraction Engine [7], GETT [106], and libtensor are examples of systems and techniques that transform (higher-dimensional) tensor contractions into dense matrix multiplications by transposing tensor operands. TBLIS [70] and InTensLi [64], by contrast, avoid explicit transpositions by computing tensor contractions in-place. Many dense tensor compilers have been developed for deep learning as well, including Tensor Comprehensions [115] and TVM [23]. Furthermore, Halide [87] showed how image processing pipelines can be optimized by having their semantics (i.e., *algorithms*) and optimization strategies (i.e., *schedules*) be separately specified. However, other than those that are also mentioned in Section 8.1, all of these systems and techniques work exclusively with data stored as dense arrays.

8.4 Data Structure Operation Synthesis

There exists a separate line of works on synthesizing data structure operations from declarative specifications, which is reminiscent of our approach for supporting pointer-based sparse tensor formats. Many techniques have been proposed for synthesizing

imperative programs that modify pointer-based data structures like AVL trees and linked lists, given either user-specified invariants [62, 86] or graphical specifications of the desired programs' inputs and outputs [99]. Other techniques have also been proposed for synthesizing functional programs from declarative specifications, including programs that process and manipulate pointer-based data structures [55, 84]. None of these techniques consider block data structures like C-trees that store chunks of elements in each node, and they do not generate parallel code. In addition, Rayside et al. [89] show how Java iterators can be synthesized for pointer-based data structures given specifications written in relational logic, though their technique does not generate map functions or any other code to actually compute with elements stored in these data structures.

8.5 Query Language Compilation

There also exists a separate line of works [22, 77, 48, 60] on generating efficient code for query languages such as SQL, which our attribute query language resembles. (Attribute queries are analogous to GROUP BY queries on a table that stores the coordinates of a tensor's nonzeros.) In particular, HorseIR [22] lowers SQL queries to an array-based intermediate representation that is then optimized and compiled to efficient code. EmptyHeaded [2] is a graph processing framework that generates efficient code to compute graph queries expressed in a Datalog-like language. Furthermore, our approach to optimizing attribute queries is reminiscent of query rewriting systems in certain relational database systems like Starburst [83, 82]. All these techniques are designed for queries that may perform complex joins and aggregate data of arbitrary types. By contrast, our attribute queries are limited to aggregating tensor coordinates, which are integers. This enables our technique to lower and optimize attribute queries in ways that would be invalid for aggregations over other arbitrary data types.

Chapter 9

Conclusion and Future Work

We have seen how to build a compiler that supports efficiently computing on sparse tensors stored in disparate formats. In particular, I have shown how a wide range of sparse tensor formats can all be expressed as compositions of level formats that use different array-based or pointer-based data structures to store coordinates of nonzeros. I have also shown how array-based level formats can be precisely defined by implementing a common set of abstract interfaces that capture how their underlying arrays can be efficiently accessed or constructed. I have further shown how pointer-based level formats can be precisely defined by specifying how stored nonzeros are distributed within nodes and by specifying how those nodes are linked together. Finally, I have shown how a compiler can use these specifications of level formats to generate efficient code for computing on sparse tensors that may be stored in arbitrary formats.

Our approach makes it possible to build a compiler that supports a wide range of formats without sacrificing the maintainability of the compiler. As we have seen, our abstractions can be implemented for each format completely independently of other formats, which makes it practical to add support for a large number of formats. Additionally, by decoupling specifications of sparse tensor formats from the code generation mechanism, our technique enables users to extend the compiler to support custom formats without having to modify the code generator itself. In these ways, our technique makes it possible to build general-purpose systems that can effectively support a broad range of real-world applications in

disparate domains, which may have to efficiently perform dissimilar computations on sparse data that possess distinct characteristics.

While our technique is a good first step towards democratizing sparse computing, I believe there are still many ways in which our technique can be further improved:

Support for additional formats Though our technique supports a wide variety of sparse tensor formats, our abstractions unfortunately do not fully capture many other interesting formats that have been proposed. These include, for instance, formats for storing block matrices with variably-sized blocks [4], formats for storing symmetric sparse tensors [95], and hybrid formats that use different data structures to store subtensors with distinct sparsity structures [13, 108]. Extending our abstractions to support such formats as well would make our technique more useful for an even wider range of applications that work with different types of sparse data.

Unified abstraction for array-based and pointer-based formats Our technique uses separate sets of abstractions to represent array-based formats and pointer-based formats. This increases the compiler's complexity since separate code paths are required to emit code that operate on array-based and pointer-based formats, even when many pointer-based formats actually use arrays to store blocks of nonzeros and thus share similarities with array-based formats. Developing a unified abstraction that is able to represent both array-based and pointer-based formats can potentially alleviate this issue, resulting in a more maintainable compiler.

Support for non-CPU hardware Our technique currently only fully supports generating code for CPUs. Senanayake et al. [96] show how our technique can be extended to generate GPU code, but this extension only supports tensor formats that are expressed as compositions of the `dense` and `compressed` level formats. Further extending our technique so that it generates efficient code for computing on tensors in any format using GPUs and other specialized hardware [41, 124] would let our technique better exploit parallelism that is inherent in many real-world sparse tensor computations.

Improved support for sparse tensor assembly To support generating code that store the results of sparse tensor algebra computations in sparse tensor formats, our

technique requires users to implement level functions that capture how tensors can be efficiently constructed in those formats. However, such level functions are often significantly more complex—and thus more cumbersome for users to implement—than level functions that capture how the same formats can be efficiently accessed. Moreover, implementations of these level functions are targeted to CPUs and, again, do not straightforwardly generalize to non-CPU hardware. One potential approach for addressing these limitations, at least for array-based formats, is to instead have users declaratively specify how each element in the underlying arrays are computed, in terms of either the coordinates of nonzeros being stored or statistics about those nonzeros. Such declarative specifications can potentially be more concise and thus simpler for users to provide. At the same time, by not constraining the algorithm used to initialize the arrays, such declarative specifications can potentially let a compiler generate sparse tensor assembly code that use distinct algorithms on different hardware.

Support for in-place modification of sparse tensors While our technique generates code that assemble sparse tensors from scratch in disparate formats, our technique does not support generating code to insert nonzeros into—or delete nonzeros from—sparse tensors that have already been assembled. This is particularly important for applications that work with dynamic sparse tensors. We believe that in-place modification of tensors benefits less from a compiler-based approach, since a programmer only needs to manually implement a bounded number of optimized routines (i.e., one for inserting nonzeros and one for deleting nonzeros) in order to support efficiently in-place modifying tensors in any particular format. That said, generalizing our technique so that it can also generate efficient code for in-place modifying sparse tensors would further reduce the programmer effort needed to add support for new formats.

I believe addressing any of these limitations would constitute valuable future work.

Historically, developing and implementing support for novel sparse tensor formats have required almost Herculean effort worthy of many PhD degrees. This unfortunate reality has limited the ability of systems for sparse computing to support many formats, thus requiring users to make their data fit with the available formats. In turn, this made it difficult

to effectively apply sparse computing techniques to domains for which existing sparse programming systems and libraries are not explicitly designed. I believe our approach has the potential to flip this script by enabling users to instead develop formats that are tailored for—and that can optimize computations on—any data they might have. This can help jump-start the development of novel applications that must work with sparse data but that are not already well-supported by existing systems, including perhaps even applications that have yet to be conceived.

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

[3] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. 2016. Improving Performance of Sparse Matrix Dense Matrix Multiplication on Large-Scale Parallel Systems. *Parallel Comput.* 59, C (nov 2016), 71–96. https://doi.org/10.1016/j.parco.2016.10.001

[4] Peter Ahrens and Erik G. Boman. 2020. On Optimal Partitioning For Sparse Matrices In Variable Block Row Format. (2020). https://doi.org/10.48550/ARXIV.2005.12414

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

[6] 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

[7] 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.

[8] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN Automatic Coding System. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability (IRE-AIEE-ACM '57 (Western))*. Association for Computing Machinery, New York, NY, USA, 188âĂŞ198. https://doi.org/10.1145/1455567.1455599

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

[10] 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.

[11] 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.

[12] 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

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

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

[15] Aart JC Bik. 1996. *Compiler Support for Sparse Matrix Computations*. Ph.D. Dissertation. Leiden University.

[16] Aart J.C. Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* (jun 2022). https://doi.org/10.1145/3544559 Just Accepted.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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).

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

[23] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[24] Stephen Chou. 2018. *Unified Sparse Formats for Tensor Algebra Compilers*. S.M. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://tensor-compiler.org/files/chou-sm-thesis-taco-formats.pdf

[25] Stephen Chou and Saman Amarasinghe. 2022. Compilation of Dynamic Sparse Tensor Algebra. Conditionally accepted in *Proc. ACM Program. Lang.* 6, OOPSLA (Dec. 2022).

[26] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (Oct. 2018), 30 pages.

[27] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 823–838. https://doi.org/10.1145/3385412.3385963

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[29] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

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

[31] 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

[32] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 918–934. https://doi.org/10.1145/3314221.3314598

[33] Daniel Donenfeld, Stephen Chou, and Saman Amarasinghe. 2022. Unified Compilation for Lossless Compression and Sparse Computing. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[34] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

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

[36] 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.

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

[38] Peter Gottschling, David S. Wise, and Michael D. Adams. 2007. Representation-transparent Matrix Algorithms with Scalable Performance. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS '07)*. ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/1274971.1274989

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

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

[41] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[42] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. 2021. Compilation of Sparse Array Programming Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 128 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485505

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

[44] Intel. 2020. Intel oneAPI Math Kernel Library Developer Reference. (2020).

[45] Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., USA.

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

[47] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001. SciPy: Open source scientific tools for Python. (2001). http://www.scipy.org/

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

[49] 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.

[50] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, USA.

[51] 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

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

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

[54] 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

[55] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

[56] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. 2016. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 842–853. https://doi.org/10.1109/IPDPS.2016.117

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

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

[59] 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.

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

[61] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[62] Darya Kurilova and Derek Rayside. 2013. On the Simplicity of Synthesizing Linked Data Structure Operations. *SIGPLAN Not.* 49, 3 (oct 2013), 155–158. https://doi.org/10.1145/2637365.2517225

[63] Tobin J. Lehman and Michael J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 294–303.

[64] 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.

[65] Jiajia Li, Yuchen Ma, and Richard Vuduc. 2018. ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs. (Oct 2018). http://parti-project.org Last updated: Jan 2020.

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

[67] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri 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

[68] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

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

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

[71] 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.

[72] 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.

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

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

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

[76] 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.

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

[78] L. Page, S. Brin, R. Motwani, and T. Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html

[79] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. https://doi.org/10.1145/3448016.3457313

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

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

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

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

[84] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[85] 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.

[86] Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural Synthesis of Provably-Correct Data-Structure Manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 65 (oct 2017), 28 pages. https://doi.org/10.1145/3133889

[87] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211

[88] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. 2017. Optimizing CNNs on Multicores for Scalability, Performance and

Goodput. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/3037697.3037745

[89] Derek Rayside, Vajihollah Montaghami, Francesca Leung, Albert Yuen, Kevin Xu, and Daniel Jackson. 2012. Synthesizing Iterators from Abstraction Functions. *SIGPLAN Not.* 48, 3 (sep 2012), 31–40. https://doi.org/10.1145/2480361.2371407

[90] Karin Remington and Roldan Pozo. 1996. *NIST Sparse BLAS User's Guide*. Technical Report. Internal Report NISTIR 6744, National Institute of Standards and Technology.

[91] 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.

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

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

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

[95] Martin D. Schatz, Tze Meng Low, Robert A. van de Geijn, and Tamara G. Kolda. 2014. Exploiting Symmetry in Tensors for High Performance: Multiplication with Symmetric Tensors. *SIAM Journal on Scientific Computing* 36, 5 (2014), C453–C479. https://doi.org/10.1137/130907215 arXiv:https://doi.org/10.1137/130907215

[96] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (nov 2020), 30 pages. https://doi.org/10.1145/3428226

[97] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In *High Performance Computing*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes (Eds.). Springer International Publishing, Cham, 97–119.

[98] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (feb 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[99] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

[100] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT file formats. (2017). http://frostt.io/tensors/file-formats.html

[101] 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/

[102] 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.

[103] 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.

[104] 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.

[105] 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.

[106] Paul Springer and Paolo Bientinesi. 2018. Design of a High-Performance GEMM-like Tensor–Tensor Multiplication. *ACM Trans. Math. Softw.* 44, 3, Article 28 (jan 2018), 29 pages. https://doi.org/10.1145/3157733

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

[108] 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

[109] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: Parallel Augmented Maps. *SIGPLAN Not.* 53, 1 (feb 2018), 290–304. https://doi.org/10.1145/3200691.3178509

[110] TensorFlow Developers. 2022. Ragged tensors. (8 June 2022). https://www.tensorflow.org/guide/ragged_tensor

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

[112] The SciPy community. 2018. scipy.sparse.lil_matrix – SciPy v1.1.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html. (2018).

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

[114] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[115] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. (2018). arXiv:cs.PL/1802.04730

[116] Anand Venkat. 2016. *An Integrated Compiler and Runtime Framework for Sparse Matrix Codes*. Ph.D. Dissertation. University of Utah.

[117] 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.

[118] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Article 41, 12 pages.

[119] 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+.

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

[121] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091058

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

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

[124] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[125] Huihui Zhang, Anand Venkat, and Mary Hall. 2016. Compiler Transformation to Generate Hybrid Sparse Computations. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 34–41. https://doi.org/10.1109/IA3.2016.011